

# **Lightning Guide to Databases with Microsoft Access and SQL**

© Prof. Dr. Arturo Azcorra

August 2021 – First edition v1.3

## **PARTS IN THIS LIGHTNING GUIDE**

PART A. CREATING MY FIRST DATABASE WITH MS-ACCESS .....	1
PART B. BRIEFING ON MS-ACCESS USER INTERFACE.....	24
PART C. CONCEPTS AND INTERNALS OF DATABASES .....	92
PART D. DESIGNING MY DATABASES WITH MS-ACCESS.....	133
PART E. ENTERING, MODIFYING AND DELETING MY DATABASE DATA .	202
PART F. WRITING SQL QUERIES TO USE MY DATABASE .....	231
PART G. WRITING EXPRESSIONS .....	398
PART H. CUSTOMIZING THE APPEARANCE OF A QUERY/TABLE/FORM IN “DATASHEET VIEW”.....	439
PART I. EVOLVING MY DATABASE DESIGN.....	469
PART J. DEBUGGING MY SQL QUERIES.....	495
PART K. USEFUL DESIGN ADVICE .....	555
PART L. FIXING DATABASE ERRORS .....	669
PART M. LIST OF BUILT-IN FUNCTIONS .....	720
PART N. CONTENTS AND ACKNOWLEDGEMENTS.....	I

## PART A. CREATING MY FIRST DATABASE WITH MS-ACCESS

You may click:

- “A.1 How do I use this Lightning Guide?”
- “A.2 What version of MS-Access is this Guide for?”
- “A.3 How do I create my first database?”
- “A.4 How do I write and run my first SQL Query?”
- “A.5 How do I add a Table to my first database?”
- “A.6 How do I write and run my first Select Query with record aggregation?”
- “A.7 How do I configure my Tables in my first database?”
- “A.8 How do I write and run my first Union Query?”
- “A.9 How do I create a Relationship in my first database?”
- “A.10 How do I write and run my first Join Query?”
- “A.11 How do I write and run my first Transform Query?”

### A.1 How do I use this Lightning Guide?

If you are new to MS-Access, to databases and to SQL, I suggest you start reading from this “**Part A. Creating my first database with MS-Access**”.

If you are familiar with SQL and databases, but not with MS-Access, you probably want to start reading from “**Part B. Briefing on MS-Access user interface**” and then move to “**Part D. Designing my databases with MS-Access**”.

If you are already quite experienced with MS-Access, but not so much with database concepts, nor with SQL, you may probably want to start reading from “**Part C. Concepts and internals of databases**” and then move to “**Part F. Writing SQL Queries to use my database**”.

If you are already quite experienced with MS-Access and database concepts, and your interest is mainly on SQL and advanced features, you may probably want to start reading from “**Part F. Writing SQL Queries to use my database**”.

The objective of this **Part A** is that you get familiar with MS-Access’ user interface, with a few of its commands, with a simple database that you will create, and with SQL Queries that you will write. Very few technical concepts or detailed commands are presented here. However, with this **Part A** you will **learn by doing** the most fundamental ideas of databases.

You can read **Part A** in “**normal mode**”, by actually typing the commands and data to create your first database, or, in “**lightning mode**”, by downloading the database and just looking into it while following my indications. The same applies to **Part F**, where I will propose a number of examples of Queries that you can replicate yourself, or just run in my database of examples. For “**lightning mode**” you can download the MS-Access database with all the examples of Tables, Forms and Queries in this **Lightning**

Guide from the link:

[https://lightningguide.net/Company\\_Database.zip](https://lightningguide.net/Company_Database.zip)

Finally, I want to remark that this is **not** an academic book. The objective of this **Lightning Guide** is that you learn databases, SQL and MS-Access **as fast as possible**. Achieving this main objective requires a little flexibility in respect to academic rigor.

## A.2 What version of MS-Access is this Guide for?

For the **English** version of **MS-Access 365** of the year **2021**.

This version is **very** similar to **earlier MS-Access 365** versions, to **MS-Access 2019** and to **MS-Access 2016**, so **almost everything** of this **Lightning Guide** also applies to those versions.

**If you have changed** options and settings of your MS-Access 365, or you are using a **different** MS-Access version, then your views and command layout will have minor differences from what I am explaining here. In that case, I hope you will be experienced enough to notice the format and view differences and follow the explanation even if your screen layout/format is not exactly the same as I am indicating.

## A.3 How do I create my first database?

You may click:

- “A.3.1 What is the MS-Access user interface?”
- “A.3.2 How do I create my first database file?”
- “A.3.3 How do I create my first Table?”
- “A.3.4 How do I input my first data into my Table?”

### A.3.1 What is the MS-Access user interface?

It is very convenient that you keep the MS-Access window **maximized** (i.e., covering all screen), because if you make it smaller many commands will not show properly, and views may differ from what I am detailing.

If you want a briefing on the MS-Access user interface before attempting to create your first database and Queries, you may click “**Part B. Briefing on MS-Access user interface**” and then come back to this **Part A**.

### A.3.2 How do I create my first database file?

Open MS-Access, and check if you see either the “**New blank database**” or the “ New” button at the window’s top left.




If you see the “ New” button, you click on “” and you will now see the “**New blank database**” button.

Once you see the “**New blank database**” button, you click on it and you get a file selector sub-window. Choose the **folder** you want for your new database file by clicking on the corresponding directories of the file selector.

The default database file name is “Database1.accb” or something similar. Click on the

default file name to select it and edit the file name to become “Test Database.accdb” (I show the text to type-in between quotes, but you **should not** type-in the quotes). It is essential that you **do not** change the file extension (i.e., the suffix of the file name after the period), that **must** remain to be “.accdb”. Click on “OK”, and **you have created your first database file: congratulations!**

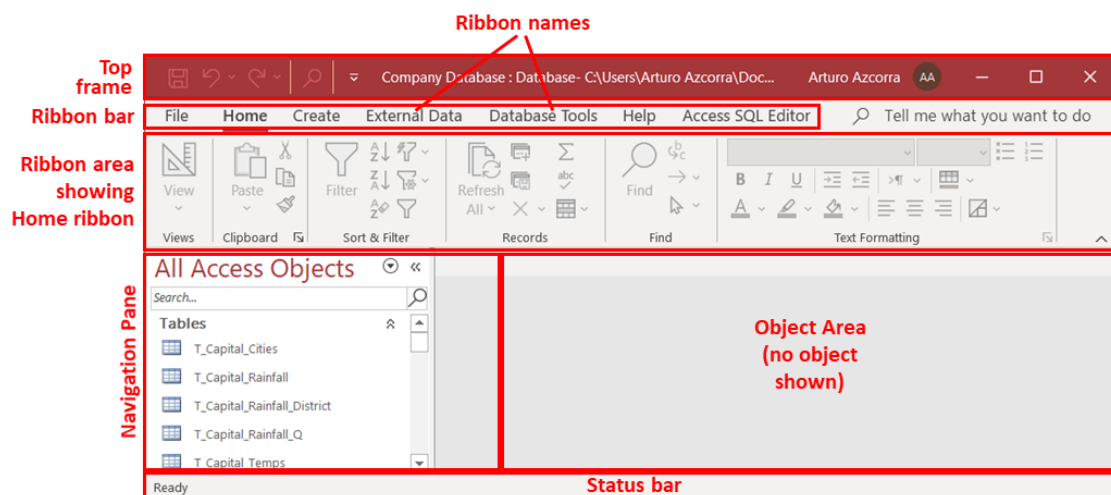
Take note of the **folder path and file name** that you used because you will want to work later over this file, following the indications of other examples in this **Lightning Guide**.

Now click on the “**Home**” Ribbon name, placed on the top left corner of the MS-Access window. This will make the **Home** Ribbon (a “Ribbon” is a toolbar placed towards the top of the MS-Access window) appear. Click on the **pin** “” icon placed at the bottom right corner of the Ribbon. The **pin** “” icon will change to the **menu** “” icon and the **Ribbon area** will be shown **permanently** instead of being automatically hidden. A **permanent Ribbon area** will be much more convenient for you until you are more familiar with MS-Access.

### A.3.3 How do I create my first Table?

If you just read A.3.2, you should have an opened database file in MS-Access and a permanent Ribbon area. Otherwise, click A.3.2 and follow my indications until you reach here.

When you have an **opened database file**, you should see **inside** the MS-Access window the following elements: the top window frame, in dark red color; the “**Ribbon-bar**” (a list of **Ribbon names**: “File”, “Home”, “Create”, etc.) right below the top window frame; a Ribbon (a wide toolbar with command icons) below the “Ribbon-bar”; the “**Navigation Pane**” below the Ribbon on the left side; and the “Object Area” below the Ribbon on the right side. The following screenshot highlights each of these elements:



I will now explain in a little more detail each of these user interface elements:

- The “**Ribbon-bar**”, at the top, contains a list of “**Ribbon names**”. If you click on a “Ribbon name” (except “File”), the corresponding “Ribbon” (i.e., toolbar) will be shown.
- A “**Ribbon**” (toolbar) below the “Ribbon-bar” contains many icons of commands and tools to do different actions on your database and database objects. You can decide what “Ribbon” is shown in the Ribbon area by clicking on a “**Ribbon name**”




(except **“File”**) from the **“Ribbon-bar”**.



- The **“Navigation Pane”**, on the left, lists the Table names, Query names and names of other database objects.
- The **“Object Area”** is the area where the opened database Objects (Tables, Queries, Forms, etc.) are shown. Each opened object type will have its specific type of pane, showing information and properties of the Object.



You will now create your first database Table.



Right-click on the **Table name “Table1”** (either at the left on the **“Navigation Pane”** or at the top on the **“Object Tab”**) and in the pop-up menu click on **“Design View”**. This will open a box requesting you to input the Table name. Type-in **“T\_Capital\_Cities”** (do not type any name different from this, not even translating into your language) and click on **“OK”**. I show the text to type-in between quotes, but you **should not** type-in the quotes.

In case that **“Table1”** was not shown, then click on **“Create”** from the **“Ribbon-bar”**, and then on the **Table Design**  icon inside the **“Create”** Ribbon. This will open a Table in **“Design View”**, with the temporal name **“Table1”**.

You should now see this new Table name in the **“Navigation Pane”** to the left, and its **“Table pane”** should show the Table in **“Design View”**.

Click on the cell placed right below **“Field name”** and type-in **“Capital”** into it (in case that cell has an existing text **“ID”** delete it and type-in **“Capital”**). Now click on the cell to the right of **“Capital”**. You will see the **drop-down menu**  icon at the rightmost side of the cell. Click on the **drop-down menu**  icon and then click on **Short Text** from the drop-down menu.

Now click on the cell below **“Capital”** and type-in **“State\_Province”**. Now click on the cell to the right of **“State\_Province”**. You will see again the **drop-down menu**  icon at the rightmost side of the cell. Click on the **drop-down menu**  icon and then click on **Short Text** from the drop-down menu.

Now click on the cell below **“State\_Province”** and type-in **“Country”**. Now click on the cell to the right of **“Country”**. You will see again the drop-down menu  icon at the rightmost side of the cell. Click on the **drop-down menu**  icon and then click on **Short Text** from the drop-down menu.

Right-click on the **Table name “Table1”** (either in the left **“Navigation Pane”** or in the Object tab) and in the pop-up menu click on **“Datasheet View”**. This will open a box asking if the Table should be saved, where you should click on **“Yes”**. Then MS-Access will ask for the Table name: type-in **“T\_Capital\_Cities”**. You have created **your first database Table: congratulations!**

You will now see your Table, with its name **“T\_Capital\_Cities”** in its **“Table pane”**, and three columns named **“Capital”**, **“State\_Province”** and **“Country”**. You will also see a fourth column named **“Click to Add”** that you should ignore (**do not click on it!**). You will also see one empty row (with an asterisk **“\*”** to its left), which is the row to enter new records into a Table.

### A.3.4 How do I input my first data into my Table?

If you just read A.3.3, you should have an opened database file with a Table. Otherwise, click A.3.3 and follow my indications until you reach here.

Click on the **first cell** to the right of the **box with an asterisk “\*”** and type-in “Beijing” (I show the text to type-in between quotes, but you **should not** type-in the quotes). Click on the cell, **in this same row**, below “Country” and type-in “China”.

Now click on the **first cell** to the right of the box with an asterisk “\*” and type-in “Brasilia”. Click on the cell, **in this same row**, below “Country” and type-in “Brazil”.

Repeat the same process **for all the rows shown in the Table below**. This is, typing-in **for each row** of the Table, the text contained in **all the cells in each row**. **Avoid** typing-in information by **columns** (e.g., typing-in all the “Capital” city names, and then all the “State\_province” names, and then all “Country” names), because MS-Access manages **each Table row** as single entity (**a record**). The values you should type-in for the cells **in each Table row** are shown in the following Table:

T_Capital_Cities		
Capital	State_Province	Country
Beijing		China
Brasilia		Brazil
Buenos Aires		Argentina
Madrid	Madrid	Spain
Washington	District of Columbia	United States

You should now see in your Table the same information as listed in the Table above. If any cell does not show exactly the same information as the Table above, click on that cell(s) and edit the information until it is exactly the same as shown above.

Now right-click on the **tab heading** (showing the Table name “T\_Capital\_Cities”) and in the pop-up menu click on “**Save**”. Next, right-click again on the **tab heading** (showing the Table name “T\_Capital\_Cities”) and in the pop-up menu click on “**Close**”. **You have input your first data into your first Table: congratulations!**

You probably were tempted to input in your database Table other countries or other information, different from the one in the table above. For example, you may want to include the capital city in your country, or some of the state/province that are blank. I strongly recommend you **do not do it**, and just type-in **exactly the information above**. The reason is that I will use this Table for other examples along the **Lightning Guide**, and if you do not have **exactly the same** information in your database, your results will differ from mine, and the examples will be very difficult to follow. If you have already typed-in different data, I suggest you go back cell by cell and modify it until your Table is exactly the same as the Table above.

## A.4 How do I write and run my first SQL Query?


You may click:

- “A.4.1 How do I write my first SQL Query?”

- “A.4.2 How do I run my first SQL Query?”
- “A.4.3 What does it mean “case insensitive”?”

### A.4.1 How do I write my first SQL Query?

If you just read A.3, you should have an opened database file with a Table having data. Otherwise, click A.3 and follow my indications until you reach here.

Click on “**Create**” from the “Ribbon-bar” (placed right below the top window frame). This will cause the “**Create**” Ribbon (toolbar) to be shown. Now click on the **Query Design** “” icon inside the “**Create**” Ribbon. This will open a “Query pane” called “Query1” or something similar. The “Query pane” is placed inside the “Object Area” (click B.2), where the “Table pane” was shown in the previous sections. The “Query pane” has a **tab** at its top with the Query name “Query1”.

In addition to having created a “Query pane”, MS-Access has opened a box with a Table list and buttons “**Add**” and “**Close**”. Click on its “**Close**” button. Now, right-click on the **tab** of the “Query pane” (showing the Query name “Query1”) and in the pop-up menu click on “**SQL View**”. The “Query pane” is changed now to “**SQL View**”, which is all blank, with the text “SELECT ;” in its top left corner. You should click inside the “Query pane” (the sub-window) and type-in (or copy/paste) the following SQL code<sup>1</sup>:

```
SELECT "Capital of: " & Enter_Country AS Srch_Country, Capital AS Cap_City
FROM T_Capital_Cities
WHERE Country = Enter_Country;
```

Make sure what you see in the “Query pane” is exactly the code above, and if not, edit until it is exactly the same.

Right-click on the **tab** of the “Query pane” (showing the Query name “Query1”) and in the pop-up menu click on “**Close**”. MS-Access will ask you if you want to save the changes in the Query, and you should click “**Yes**”. Then MS-Access will ask for the Query name: type-in “A\_Capital\_City” (I show the text to type-in between quotes, but you **should not** type-in the quotes). **You have created your first SQL Query: congratulations!**

When you run this Query, it will display a record-list having in the **first** column (named “**Srch\_Country**”) the value that you type-in for the **parameter** “**Enter\_Country**” **prefixed** by the text string “**Capital\_of:** ”. It will also select the record (row) from the Table “**T\_Capital\_Cities**” whose value of “**Country**” is equal to the value of “**Enter\_Country**” and will display the value of the field “**Capital**” **from that specific record** in the **second** column (named “**Cap\_City**”). The value of the variable “**Enter\_Country**” is **undefined** and therefore MS-Access considers it a **parameter** (click F.12). MS-Access requests that you type-in the value of “**Enter\_Country**” each time that you run this Query.

Notice that in the SQL code above the ampersand “**&**” operator performs text string concatenation.

If you want a full description of a **Select Query**, you may click “*F.7 What is a Select operation and how do I write it?*”.

---



<sup>1</sup> This is the Query “**A\_Capital\_City**” from file “**Company\_Database.accdb**”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “F.11.2 What are the SQL color codes used in this Guide?”.

### A.4.2 How do I run my first SQL Query?

If you just read A.4.1, you should have an opened database with a Table and a Query. Otherwise, click A.4.1 and follow my indications until you reach here.

Double-click on the Query **name** “A\_Capital\_City” placed in the “**Navigation Pane**” (click B.4). A pop-up box will appear, requesting you to type-in the value of the parameter “**Enter\_Country**”. Type-in one of the countries in your Table (e.g., “United States”, always **without** the quotes) and press the “**Enter**” key (or click on the “**OK**” button). The Query will return the value of “**Enter\_Country**”, **and its correct Capital city. You have run your first SQL Query: congratulations!**

If you want to run the Query again, click on “**Home**” on the “**Ribbon-bar**”, and then on the **Refresh All**  icon, and the pop-up box to type-in “**Enter\_Country**” will be shown again. Each time you click on the **Refresh All**  icon the Query will be run.

Notice that if you do not type-in one of the countries in your Table “**T\_Capital\_Cities**”, or the country is mistyped, the Query will return a blank cell under “**Capital**”. If you type-in “Spain” you should get the following result:

A_Capital_City	
Srch_Country	Cap_City
Capital of: Spain	Madrid

Check this output record-list against my explanation above and the SQL code, to see the effect of each SQL clause and operator. Notice that in MS-Access results you will **not** see the colored fonts: I have added the colors above to help you relate the Query code with the Query output.

Notice also that the “**AS**” clauses determine the **name** of each **output** column (i.e., **output** field).

Once you are done, close this “Query pane” by right-clicking on its **tab** and in the pop-up menu click on “**Close**”.

### A.4.3 What does it mean “case insensitive”?

A “**case insensitive**” program is the one that **treats the same** an **upper-case** letter and its corresponding **lower-case** letter. MS-Access is **case insensitive**, which means that when **comparing** text strings and when **ordering** text strings, it will treat the same **upper-case** letters and **lower-case** letters.

If you run the Query from the previous section several times, and you **type-in** “SPAIN”, “spain” or “SpAiN” (or other case combinations), you will always get the capital of Spain, because MS-Access will consider all the strings as being equal.


## A.5 How do I add a Table to my first database?


You may click:



- “A.5.1 How do I add the Table “T\_Capital\_Rainfall\_Q” to my first database?”
- “A.5.2 How do I input data into the Table “T\_Capital\_Rainfall\_Q”?”



### A.5.1 How do I add the Table “T\_Capital\_Rainfall\_Q” to my first database?

If you just read A.3, you should have an opened database with a Table. Otherwise, click A.3, follow my indications, and then return here (you return by simultaneously pressing the “Alt” and “←” keys).

Click on “**Create**” from the “Ribbon-bar”, and then on the **Table Design**  icon inside the Ribbon. This will open a Table in “**Design View**”, with the temporal name “Table1”.

Click on the cell placed right below “Field name” and type-in “Capital” into it (in case that cell has an existing text “Id” delete it and type-in “Capital”). I show the text to type-in between quotes, but you **should not** type-in the quotes. Now click on the cell to the right of “Capital”. Click on the drop-down menu  icon at the rightmost side of the cell and click on **Short Text** from the drop-down menu.

Now click on the cell below “Capital” and type-in “Cal\_Year”. Then click on the cell to the right of “Cal\_Year”. Click on the drop-down menu  icon at the rightmost side of the cell and click on “**Number**” from the drop-down menu. Next, find a row called “**Field Size**”. It is one of the field properties, placed in the bottom part of the MS-Access window, in a **tab** called “**General**”. Click on the drop-down menu  icon at the rightmost part of the row called “**Field Size**” and click on “**Long Integer**” from the drop-down menu.

Now click on the cell below “Cal\_Year” and type-in “Quart”. Then click on the cell to the right of “Quart”. Click on the drop-down menu  icon at the rightmost side of the cell and click on “**Number**” from the drop-down menu. Next, find a row called “**Field Size**”. It is one of the field properties, placed at the bottom part of the MS-Access window, in a **tab** called “**General**”. Click on the drop-down menu  icon at the rightmost part of the row called “**Field Size**” and click on “**Double**” from the drop-down menu.

Right-click on the **Table name** “Table1”, either in the left “**Navigation Pane**” or in the “Table pane” **tab**, and in the pop-up menu click on “**Datasheet View**”. This will open a box requesting you to save the Table, where you should click “**Yes**”. Then MS-Access will ask for the Table name: type-in “T\_Capital\_Rainfall\_Q” (always **without** the quotes). You are done creating the **second** Table in your database.

### A.5.2 How do I input data into the Table “T\_Capital\_Rainfall\_Q”?

If you just read A.5.1, you should have an opened database with **two** Tables. Otherwise, click A.5.1 and follow my indications until you reach here.

Click on the **first cell** to the right of the **box with an asterisk** “\*” and type-in “Beijing” (I show the text to type-in between quotes, but you **should not** type-in the quotes). Click

on the cell, **in this same line**, below “Cal\_Year” and type-in “2018”. Click on the cell, **in this same line**, below “Quart” and type-in “Q1” and do the same for the last cell **in this same line**, typing-in “0” into it.

Now click on the **first** cell to the right of the box with an asterisk “\*” and type-in “Beijing”. Repeat the same process **for all the cells in this line**.

Repeat the same process **for all the rows shown in the Table below**. This is, typing-in **for each row** of the Table, the text contained in **all the cells in each table row**. **Avoid** typing-in information by **columns** (i.e., **do not** type-in first all the Capital names, and then all the State or Province names, and then all Country names), because MS-Access **manages each Table row** as single entity (**a record**), so it is better if you input data by rows at this stage. The values you should type-in for the cells **in each Table row** are shown in this Table:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

You should now see in your Table the same information as listed in the Table above. If any cell does not show **exactly** the same information as the table above, click on that cell(s) and edit the information until it is exactly the same as shown above.

Finally, right-click on the Table name “**T\_Capital\_Rainfall\_Q**” in the **tab** heading and in the pop-up menu click on “**Save**”. You have created and populated the second Table in your database.


## **A.6 How do I write and run my first Select Query with record aggregation?**

You may click:

- “A.6.1 How do I write my first Select Query with record aggregation?”
- “A.6.2 How do I run my first Select Query with record aggregation?”

### **A.6.1 How do I write my first Select Query with record aggregation?**

If you just read A.3 and A.5, you should have an opened database with two Tables. Otherwise, click A.3 and A.5, follow my indications there, and then return here (you return by simultaneously pressing the “**Alt**” and “**←**” keys).

Click on “**Create**” (placed right below the top window frame). This will cause the “**Create**” Ribbon (toolbar) to be shown. Now click on the **Query Design** “” icon inside



the “**Create**” Ribbon. This will open a “Query pane” called “Query2” or something similar. The “Query pane” is like a sub-window and has a **tab** at its top with the Query name “Query2”.

In addition to having created a “Query pane”, MS-Access has opened a box with a Table list and buttons “**Add**” and “**Close**”. Click on its “**Close**” button. Now, right-click on the Query **tab** and in the pop-up menu click on “**SQL View**”. The “Query pane” is changed now to “**SQL View**”, which is all blank, with the text “SELECT;” in its top left corner. You should click inside the “Query pane” (the sub-window) and type-in (or copy/paste) the following SQL code<sup>2</sup>:

```
SELECT Capital AS Cap_City, Cal_Year+0 AS C_Year
, Sum(Quart_Rainfall) AS Yearly_Rainfall
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year+0
ORDER BY Capital, Cal_Year+0 ;
```

Make sure what you see in the “Query pane” is exactly the code above, and if not, edit until it is exactly the same.

Right-click on the Query **tab** and in the pop-up menu click on “**Close**”. MS-Access will ask you if you want to save the changes in the Query, and you should click on “**Yes**”. Then MS-Access will ask for the Query name: type-in “A\_Yearly\_Rainfall” (always **without** the quotes).

This Query will select **all** the records (rows) from the Table “**T\_Capital\_Rainfall\_Q**”. It will **classify** the records in **disjoint groups** that have the same **value** of the “**GROUP BY**” **expressions** “**Capital**” and “**Cal\_Year+0**”. For **each** such **group** of records, it will produce **one output record**, with the value of the “**GROUP BY**” **expressions** “**Capital**”, “**Cal\_Year+0**”, and also, the **sum** of the values of the **input field** “**Quart\_Rainfall**” from **all** the records **in each group**. This last field will be named “**Yearly\_Rainfall**”. Finally, the **output** record-list will be ordered by the values of the “**GROUP BY**” **expression** “**Capital**”, and the records where the values of “**Capital**” are the same, will be ordered by the values of the “**GROUP BY**” **expression** “**Cal\_Year+0**”.

Once you are done, close this “Query pane” by right-clicking on its **tab** and in the pop-up menu click on “**Close**”.

Notice also that I have used “**Cal\_Year+0**” as a “**GROUP BY**” **expression**, instead of just “**Cal\_Year**”: I do this to **highlight** that you can use **any expression** involving **input** field names, and **not just** the **input** field names **as such**.

If you want a full description of a **Select** Query (including the “**GROUP BY**” aggregation), you may click “*F.7 What is a Select operation and how do I write it?*”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “*F.11.2 What are the SQL color codes used in this Guide?*”.

### **A.6.2 How do I run my first Select Query with record aggregation?**

If you just read A.6.1, you should have an opened database with **two** Tables and a **Select** Query with record aggregation. Otherwise, click A.6.1 and follow my indications until

---

<sup>2</sup> This is the Query “A\_Yearly\_Rainfall” from file “Company\_Database.accdb”.

you reach here.

Double-click on “A Yearly Rainfall” placed in the “**Navigation Pane**” (click *B.4*). MS-Access will now open a “Query pane” in “**Datasheet View**”, with a tab at its top named “A Yearly Rainfall”. In this “Query pane” you should see the following **output** record-list:

A_Yearly_Rainfall		
Cap_City	C_Year	Yearly_Rainfall
Beijing	2018	28.8
Washington	2018	32.76

Notice that you will **not** see the colored fonts: I have added the colors above to help you relate the Query code with the Query output.

You can see that the Query has **assigned each** row (record) to **one** disjoint **group** such that the records in each **group** have the same **values** in the “**GROUP BY**” expressions “**Capital**” and “**Cal\_Year+0**”, and out of **each group** it produces **one output** record having one field that is the **sum** of the values of the **column** (called “**input field**”) “**Quart\_Rainfall**” in all the **rows** (called **records**) in **each group**, and naming that added result as “**Yearly\_Rainfall**”. You can see that the **grouping** of rows (records) comes from the “**GROUP BY**” expressions in the SQL code.

Notice that the “**AS**” clauses determine the **name** of the **output** columns.

Notice also that the **output** rows from the Query are **ordered** by the values of the “**GROUP BY**” expression “**Capital**”, and the records where the values of “**Capital**” are the same, are ordered by the values of the “**GROUP BY**” expression “**Cal\_Year+0**”. This comes from the “**ORDER BY**” clause in the SQL code.

## A.7 How do I configure my Tables in my first database?

You may click:

- “A.7.1 How do I configure my Table “T Capital Cities”?”
- “A.7.2 How do I configure my Table “T Capital Rainfall Q”?”

### A.7.1 How do I configure my Table “T Capital Cities”?

You may click:

- “A.7.1.1 How do I configure the Primary Key field in Table “T Capital Cities”?”
- “A.7.1.2 How do I configure no zero-length in all my text fields?”
- “A.7.1.3 How do I check that the “Required” property really works?”
- “A.7.1.4 How do I check that the Key field really works?”

#### A.7.1.1 How do I configure the Primary Key field in Table “T Capital Cities”?



If you just read A.3 and A.5, you should have an opened database with two Tables. Otherwise, click A.3 and A.5, follow my indications there, and then return here (you



return by simultaneously pressing the “Alt” and “←” keys).

The **Key** field is the one that “identifies” each record in the Table. The **Key** field must have a unique (no duplicate) value in the Table. If you want to know more about the **Key** fields, you may click “C.10 *What are the Table Key(s) and how should I handle them?*” and “D.6 *How do I configure the Primary Key field(s) of a Table?*”. If you think about it, the field “Capital” seems the **Primary Key** field of this Table, so let us configure it.

First, open the Table in “**Design View**”. To do it, right-click on “T\_Capital\_Cities” from the “**Navigation Pane**” (click B.4) and in the pop-up menu click on “**Design View**”.

Now select the row (field) “Capital” by clicking on the small gray box to the left of “Capital”. It should now be highlighted (shaded) to indicate it has been selected (if not, try again). You now click on “**Design**” from the “**Ribbon-bar**” (below the “**Table Tools**” contextual label), and then click on the **Primary Key** “” icon inside the Ribbon (toolbar). You should now see a key “” icon right to the left of “Capital”: this shows that you have successfully configured “Capital” as the **Key** field of this Table.

Notice that when you configure a field as a **Key** field, MS-Access automatically configures it as “**Required=Yes**”, which means that this field **cannot** be empty (i.e., cannot be **Null**). Also, if you configure a field as the **only Key** field in the Table, MS-Access automatically configures its “**Indexed**” property as “**Yes (No duplicates)**”, which means that this field will be indexed, and also, that MS-Access will not allow you to enter any duplicate (i.e., already existing) value in this field. If you want to know more about indexing you may click “C.8 *What is indexing?*”.

#### **A.7.1.2 How do I configure no zero-length in all my text fields?**

If you just read A.7.1.1, you should have an opened database with **two** Tables. Otherwise, click A.7.1.1 and follow my indications until you reach here.

Let us now configure no “zero-length” in **all Short Text** fields. This is a very good practice, because this prevents all-blank strings in your text fields. All-blank strings usually create errors, so they are undesirable.

To configure this, click on the field “Capital”. Find the row “**Allow Zero Length**” in the field properties, placed at the bottom of the “Table pane”, in the **tab “General**”. Click at the rightmost side of the row and click on “**No**” from the drop-down menu. Do the same for the fields “State\_Province” and “Country”.

#### **A.7.1.3 How do I check that the “Required” property really works?**

If you just read A.7.1.2, you should have an opened database with **two** configured Tables. Otherwise, click A.7.1.2 and follow my indications until you reach here.

When you configured the field “Capital” as a **Key** field, MS-Access automatically configured it as “**Required=Yes**”. This implies that MS-Access prevents you from mistakenly introducing a record without a value in this field. Let us check that it works.

Right-click on the name “T\_Capital\_Cities”, either at the “**Navigation Pane**” (click B.4), or in the **tab** below the Ribbon, and in the pop-up menu click on “**Datasheet View**”.

Click now in the last row of the column “Country”, type-in “Canada” and then click on

“Washington” above (this will introduce the record you are editing in the last row). MS-Access will **not** introduce the “Canada” record in the Table because its **required** field “Capital” is blank. MS-Access will therefore show the error message:

*“You must enter a value in the 'T\_Capital\_Cities.Capital' field.”*

This is the advantage of configuring a field as “**Required=Yes**”: MS-Access now prevents the introduction of wrong records without a value in this field.

Click on “**OK**” to remove the error message, and then, press the “**Esc**” key, and the **erroneous** record you were trying to introduce will be removed.

Now, go back to “**Datasheet View**”: right-click on the name “T\_Capital\_Cities”, either at the “**Navigation Pane**” (click *B.4*), or in the **tab** below the Ribbon, and in the pop-up menu click on “**Datasheet View**”.

#### **A.7.1.4 How do I check that the Key field really works?**

If you just read *A.7.1.3*, you should have an opened database with **two** Tables. Otherwise, click *A.7.1.3* and follow my indications until you reach here.

You should have the Table “T\_Capital\_Cities” in “**Datasheet View**” (if not, see the paragraph above).

When you configured the field “Capital” as the only **Key** field, MS-Access automatically configured it as indexed with no duplicates. This implies that MS-Access prevents you from mistakenly introducing a duplicate value in this field. Let us check that it works.

Now click on the last row of the column “Capital”, type-in “Beijing” and in the other cell of this row type-in “Any\_country” (always **without** the quotes). Then click on “Washington” above (this should introduce the record you are editing in the last row). MS-Access will **not** insert the record into the Table because it contains a duplicate value in the **Key** field in respect to another record already in the Table and will show the error message:

*“The changes you requested to the table were not successful because they would create duplicate values in the index, primary key, or relationship. Change the data in the field or fields that contain duplicate data, remove the index, or redefine the index to permit duplicate entries and try again.”*

benefits of having configured the **Key** field: MS-Access now prevents the introduction of wrong duplicate records in the Table.

Click on “**OK**” to remove the error message, and then, press the “**Esc**” key, and the erroneous record you were trying to introduce will be removed.

#### **A.7.2 How do I configure my Table “T Capital Rainfall Q”?**

You may click:

- “A.7.2.1 How do I configure the Primary Key fields in Table “T Capital Rainfall Q”?”
- “A.7.2.2 How do I configure “Allow Zero Length=No” in all my text fields?”
- “A.7.2.3 How do I configure a field validation rule in my “Quart” field?”

- “A.7.2.4 How do I check that the composite Key fields really work?”
- “A.7.2.5 How do I check that the field validation rule for the field “Quart” really works?”

### A.7.2.1 How do I configure the Primary Key fields in Table “T Capital Rainfall Q”?



If you just read A.3 and A.5, you should have an opened database with two Tables. Otherwise, click A.3 and A.5, follow my indications there, and then return here (you return by simultaneously pressing the “Alt” and “←” keys).

The **Primary Key** are the field(s) that “identify” each record in each Table. The **Primary Key field(s)** must have a unique (no duplicate) **combined** value in the Table. If you want to know more about the **Primary Key field(s)**, you may click “C.10 What are the Table Key(s) and how should I handle them?” and “D.6 How do I configure the Primary Key field(s) of a Table?”. I do not think the field “Capital” **alone** is the **Primary Key**, because you can have several records with the same “Capital” value. The same happens for **all** the other fields in the Table: **none** of them has unique values. However, if you think about it, what should be **unique** is the **combined** value of “Capital”, “Cal\_Year” and “Quart”. This is what identifies each valid measurement record in the Table. Each of these unique records will have the corresponding value in the field “Quart\_Rainfall”, which represents the amount of rain that city got in that year and quarter.

Let us see now how to configure **these three fields** as the **Primary Key** of this Table.

First, open the Table in “**Design View**”. To do it, right-click on “T\_Capital\_Rainfall\_Q” on the “**Navigation Pane**” (click B.4) and in the pop-up menu click on “**Design View**”.

Now select the row (field) “Capital” by clicking on the small gray box to the left of “Capital”. This row should now be highlighted (shaded) to indicate it has been selected (if not, try again). You now “**Ctrl+click**” (i.e., do a click **while** holding down the “**Ctrl**”<sup>3</sup> key in the keyboard) on the small gray box to the left of “Cal\_Year”. This row should now be highlighted (shaded) **in addition** to the “Capital” row. You now “**Ctrl+click**” on the small gray box to the left of “Quart”. This row should now be highlighted (shaded) **in addition** to the “Capital” and “Cal\_Year” rows.

You now click on “**Table Tools -Design**” on the “**Ribbon-bar**”, and then click on the **Primary Key**  icon inside the Ribbon (toolbar). You should now see a key  icon right to the left of “Capital”, “Cal\_Year” and “Quart”: this shows you have successfully configured the fields “Capital”, “Cal\_Year” and “Quart” as the **Primary Key** this Table.

Notice that when you configure the **Primary Key field(s)**, MS-Access **automatically** configures **all of them** as “**Required=Yes**”, which means the field cannot be empty (i.e., cannot be **Null**). Therefore, these three fields are now configured as “**Required=Yes**”. MS-Access also **automatically** configures a **composite index without duplicates** over the **Primary Key field(s)**.

---

<sup>3</sup> The “**Ctrl**” key is called the “**Control**” key and is usually labeled “**Ctrl**” in keyboards.

### A.7.2.2 How do I configure “Allow Zero Length=No” in all my text fields?

If you just read A.7.2.1, you should have an opened database with **two** Tables. Otherwise, click A.7.2.1 and follow my indications until you reach here.

Let us now configure “**Allow Zero Length=No**” in all **Short Text** fields. This is a very good practice, because this prevents zero-length strings in your text fields. Zero-length strings usually create errors, so they are undesirable.

To configure this, click on the field “Capital”. Find the row “**Allow Zero Length**” in the field properties, placed at the bottom of the “Table pane”, in the tab “**General**”. Click at the rightmost side of the row and click on “**No**” from the drop-down menu. Do the same for the field “Quart”.

### A.7.2.3 How do I configure a field validation rule in my “Quart” field?

If you just read A.7.2.2, you should have an opened database with **two** configured Tables. Otherwise, click A.7.2.2 and follow my indications until you reach here.

Notice that the field “Quart” should **only** contain the values “Q1”, “Q2”, “Q3” or “Q4”<sup>4</sup>. It would be very nice if MS-Access could check this, and actually there is a specific property to do this check. Click on the field “**Quart**”. Find the row “**Validation Rule**” in the field properties, placed at the bottom of the “Table pane”, in the **tab “General**”. Click on the cell in that row, that is placed to the right of the cell that says “**Validation Rule**”. Now, type-in (or copy/paste) in that cell **exactly** the following:

```
[Quart] In ("Q1";"Q2";"Q3";"Q4")
```

As you may guess, this field validation rule implies that the value of “Quart” should be contained in that list of values (or be **Null**)

You now go back to “**Datasheet View**”: right-click on the **name** “T\_Capital\_Rainfall\_Q”, either at the “**Navigation Pane**” (click B.4), or in the Table **tab** below the Ribbon, and in the pop-up menu click on “**Datasheet View**”.

### A.7.2.4 How do I check that the composite Key fields really work?

If you just read A.7.2.3, you should have an opened database with **two** configured Tables. Otherwise, click A.7.2.3 and follow my indications until you reach here.

You should have the Table “T\_Capital\_Rainfall\_Q” in “**Datasheet View**” (if not, see the paragraph above).

Click now in the last row of the column “Capital”, type-in “Washington”. Then click on the cell to its right and type-in “2018”. Then click on the cell to its right and type-in “Q4”. Then click on the cell to its right and type-in whatever number you want. As you may see, this is a record with a duplicate combined value in the fields “Capital”, “Cal\_Year” and “Quart”.

Now you click on “Beijing” above (this will introduce the record you are editing in the last row). However, MS-Access will **not** insert this record in the Table because it has a

---

<sup>4</sup> You could of course have thought of over values, like “Q\_1”, or “Quarter\_one” or “First\_Quarter”, etc. What is important is that you want only **four values** in this field, corresponding to the way you have named the four quarters.

duplicate value in the **Key** fields in respect to another record already in the Table. MS-Access will show the error message:

*“The changes you requested to the table were not successful because they would create duplicate values in the index, primary key, or relationship. Change the data in the field or fields that contain duplicate data, remove the index, or redefine the index to permit duplicate entries and try again.”*

This is one of the benefits of having configured the **Primary Key** fields: MS-Access now prevents the introduction of **duplicate** (i.e., wrong) records in the Table.

Click on **“OK”** to remove the error message, and then, press the **“Esc”**<sup>5</sup> key, and the **erroneous** record you were trying to introduce will be removed.

#### **A.7.2.5 How do I check that the field validation rule for the field **“Quart”** really works?**

If you just read A.7.2.4, you should have an opened database with **two** Tables. Otherwise, click A.7.2.4 and follow my indications until you reach here.

You should have the Table **“T\_Capital\_Rainfall\_Q”** in **“Datasheet View”** (if not see the paragraph above).

Click now in the last row of the column **“Quart”** and type-in **“Q5”**. Then click on the cell to its left. This tells MS-Access you finished inputting the value of the cell **“Quart”**, and therefore, it will check the field validation rule. MS-Access will not introduce this field value in the editing record because entering **“Q5”** does not satisfy the field validation rule, and will show the error message:

*“One or more values are prohibited by the validation rule '[Quart] In ("Q1","Q2","Q3","Q4")' set for 'T\_Capital\_Rainfall\_Q.Quart'. Enter a value that the expression for this field can accept.”*

This is the advantage of configuring a field validation rule: MS-Access now prevents the introduction of records with a wrong value in this field.

Click on **“OK”** to remove the error message, and then, press the **“Esc”**<sup>6</sup> key, and the new record you were trying to introduce will be removed.

### **A.8 How do I write and run my first Union Query?**

You may click:

- **“A.8.1 How do I write my first Union Query?”**
- **“A.8.2 How do I run my first Union Query?”**


#### **A.8.1 How do I write my first Union Query?**

If you just read A.3 and A.5, you should have an opened database with two Tables. Otherwise, click A.3 and A.5, follow my indications there, and then return here (you return by simultaneously pressing the **“Alt”** and **“←”** keys).

---

<sup>5</sup> The **“Esc”** key is also called the **“Escape”** key.

<sup>6</sup> The **“Esc”** key is also called the **“Escape”** key.

Click on “**Create**” (placed right below the top window frame). This will cause the “**Create**” Ribbon (toolbar) to be shown. Now click on the **Query Design**  icon inside the “**Create**” Ribbon. This will open a “Query pane” called “Query3” or something similar. The “Query pane” is like a sub-window and has a **Query tab** at its top with the Query name “Query3”.

In addition to having created a “Query pane”, MS-Access has opened a box with a Table list and buttons “**Add**” and “**Close**”. Click on its “**Close**” button. Now, right-click on the **tab** at the top of the “Query pane” and in the pop-up menu click on “**SQL View**”. The “Query pane” is changed now to “**SQL View**”, which is all blank, with the text “SELECT;” in its top left corner. You now click inside the “Query pane” and type-in (or copy/paste) the following SQL code<sup>7</sup>:

```
SELECT Enter_Country_A AS Srch_Country, Capital AS Cap_City
FROM T_Capital_Cities
WHERE Country = Enter_Country_A
UNION
SELECT Enter_Country_B AS Irrelevant_1, Capital AS Irrelevant_2
FROM T_Capital_Cities
WHERE Country = Enter_Country_B
```

Make sure what you see in the “Query pane” is exactly the code above, and if not, edit until it is exactly the same.

Right-click on the **tab** of the “Query pane” and in the pop-up menu click on “**Close**”. MS-Access will ask you if you want to save the changes in the Query, and you should click “**Yes**”. Then MS-Access will ask for the Query name: type-in “A\_Capital\_of\_2\_Countries” (always **without** the quotes).

Notice this Query is like **two** copies (only changing the parameter name “Searched\_Country”) of the **Select** Query you already wrote in “*A.4.1 How do I write my first SQL Query?*”. These two **Select** operations are connected with the “**UNION**” operator. This operator just concatenates its two **input** record-lists, into a longer record-list.

This Query will therefore select the record (row) from the Table “**T\_Capital\_Cities**” whose value of “**Country**” is equal to the value of “**Enter\_Country\_A**”. It will do the same for the record whose “**Country**” is equal to “**Enter\_Country\_B**”. It will then put **both** records in the same record-list and will display the result. The values of “**Enter\_Country\_A**” and “**Enter\_Country\_B**” are undefined, and therefore, MS-Access will request that you type-in **both values** when you run this Query.

Notice that in the SQL code above the “**AS**” clauses from the **left** input-record-list of the **Union** operation determine the **name** of the **output** columns.

If you want a full description of a **Union** Query, you may click “*F.9 What is a Union operation and how do I write it?*”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “*F.11.2 What are the SQL color codes used in this Guide?*”.

---

<sup>7</sup> This is the Query “A\_Capital\_of\_2\_Countries” from file “Company\_Database.accdb”.

## A.8.2 How do I run my first Union Query?

If you just read A.8.1, you should have an opened database with **two** Tables and a **Union Query**. Otherwise, click A.8.1 and follow my indications until you reach here.

Double-click on “A\_Capital\_of\_2\_Countries” placed in the “**Navigation Pane**” (click B.4). MS-Access will now open a “Query pane” in “**Datasheet View**”, with a **tab** at its top named “A\_Capital\_of\_2\_Countries”. MS-Access will request you to type-in the value for “**Enter\_Country\_A**” and “**Enter\_Country\_B**”. If you first type-in “**China**” and then “**United States**”, you should get the following record-list (possibly with a different order):

A_Capital_of_2_Countries	
Srch_Country	Cap_City
China	Beijing
United States	Washington

Check this output record-list against my explanation above and the SQL code, to see the effect of each SQL clause and operator. Notice that you will **not** see the colored fonts: I have added the colors above to help you relate the Query code with the Query output.

Notice that the “**AS**” clauses in the **left** record-list of the **Union** operation **determine** the **name** of the **output** columns. The “**AS**” clauses in the **right** record-list of the **Union** operation have **no effect**.

Once you are done, close this “Query pane” by right-clicking on its **tab** and in the pop-up menu click on “**Close**”.

## A.9 How do I create a Relationship in my first database?

You may click:

- “A.9.1 How do I create a Relationship from “T Capital Cities” to “T Capital Rainfall Q”?”
- “A.9.2 How do I check that the Relationship really works?”

### A.9.1 How do I create a Relationship from “T Capital Cities” to “T Capital Rainfall Q”?


If you just read A.3, A.5 and A.7, you should have an opened database with two Tables properly configured. Otherwise, click A.3, A.5 and A.7, follow my indications there, and then return here (you return by simultaneously pressing the “**Alt**” and “**←**” keys).

In short, a Relationship (click C.11) implies that the records in the **slave** Table, take the values from **some** field(s) (the **slave** field(s)) from the values existing in the **corresponding** field(s) (the **master** field(s)) in an **existing record** of the **master** Table. Almost always, the master Table and the slave Table are **two different** Tables. Relationships are **extremely** useful to prevent errors, to guarantee data coherence, and to improve efficiency in the database.

In our current example database of city information, I would say that the **slave** field “Capital” from the **slave** Table “T\_Capital\_Rainfall\_Q” should only take values from the



**master** field “Capital” from the **master** Table “T\_Capital\_Cities”. I am assuming that “T\_Capital\_Cities” lists **all** the capital cities that are **relevant** for the purposes of your database (e.g., the ones where your company does business with). Consequently, you are only interested in adding records to the Table “T\_Capital\_Rainfall\_Q” such that their capital city is already in the Table “T\_Capital\_Cities”: this matches exactly the concept of a Relationship.

To establish such a Relationship, click on “**Database Tools**” from the “Ribbon-bar” at the top of the window. Then click on the **Relationships**  icon inside the Ribbon (toolbar). Now drag-and-drop the Table “T\_Capital\_Cities” from the “**Navigation Pane**” on the left, to the “**Relationships**” pane on the right. Drag-and-drop means placing the mouse over “T\_Capital\_Cities”, pressing the left mouse button **and without releasing it**, moving the mouse to place it over the “**Relationships**” pane, and finally release the left mouse button. You now do the same with the Table “T\_Capital\_Rainfall\_Q”. You should now see in the “Relationships” pane one **Table-box** for each Table, each labeled with the Table **name**, and with the Table’s field names listed inside each Table-box.

To create the Relationship, drag-and-drop the field “Capital” from the Table-box “T\_Capital\_Cities” dropping it over the field “Capital” from the Table-box “T\_Capital\_Rainfall\_Q”. A Relationship box will pop-up with the information about this Relationship. Check “**Enforce Referential Integrity**” and then check “**Cascade Update Related Fields**”. Finally, click on the “**Create**” button at the top right corner of the Relationship box. **You have created your first Relationship, congratulations!**

If everything went well, you should see a **black line connecting** the field “Capital” in the Table-box “T\_Capital\_Cities” and the field “Capital” in the Table-box “T\_Capital\_Rainfall\_Q”. The **connecting line** has a “**1**” at the endpoint of “T\_Capital\_Cities” and an infinity “**∞**” symbol at the endpoint of “T\_Capital\_Rainfall\_Q”. These symbols at each end indicate that this is a **one-to-many** Relationship (click C.11.2).

### **A.9.2 How do I check that the Relationship really works?**

If you just read A.9.1, you should have an opened database with **two** properly configured Tables and a Relationship. Otherwise, click A.9.1 and follow my indications until you reach here.

Let us now check if the Relationship really works. Right-click on “T\_Capital\_Rainfall\_Q” and in the pop-up menu click on “**Datasheet View**”. Now click on the bottommost cell of the column “Capital” and type-in “Monaco” (always without the quotes). Click on the cell to its right, and type-in “2018”. Click on the cell on its right, and type-in “Q1”. Click on the cell to its right and type-in any number you want. Finally, click on “Washington” above (this should introduce the record you are editing in the last row). MS-Access will **not** insert the record into the Table because the capital city value “Monaco” does not exist in any record of the **master** Table “T\_Capital\_Cities”, and will rather show the error message:

*“You cannot add or change a record because a related record is required in table 'T\_Capital\_Cities'.”*

This is one of the advantages of configuring a Relationship: MS-Access now prevents



the introduction of records with a value in a **slave** field that **does not exist** in any record of the **master** Table.

Click on “**OK**” to remove the error message, and then, press the “**Esc**”<sup>8</sup> key, and the new (wrong) record that you were trying to introduce will be removed.


## A.10 How do I write and run my first Join Query?

You may click:

- “A.10.1 How do I write my first Join Query?”
- “A.10.2 How do I run my first Join Query?”

### A.10.1 How do I write my first Join Query?

If you just read A.3 and A.5, you should have an opened database with two configured Tables. Otherwise, click A.3 and A.5, follow my indications there, and then return here (you return by simultaneously pressing the “**Alt**” and “**←**” keys).

Click on “**Create**” (placed right below the top window frame). This will cause the “**Create**” Ribbon (toolbar) to be shown. Now click on the **Query Design** “” icon inside the “**Create**” Ribbon. This will open a “Query pane” called “Query4” or something similar. The “Query pane” has a **tab** at its top with the Query name “Query4”.

In addition to having created a “Query pane”, MS-Access has opened a box with a Table list and buttons “**Add**” and “**Close**”. Click on its “**Close**” button. Now, right-click on the **tab** at the top of the “Query pane” and in the pop-up menu click on “**SQL View**”. The “Query pane” is changed now to “**SQL View**”, which is all blank, with the text “**SELECT;**” in its top left corner. You should click inside the “Query pane” and type-in (or copy/paste) the following SQL code<sup>9</sup>:

```
SELECT Capitals.Cap_City, St_Pr, Cntry, C_Year, Year_Rainfall
FROM
(
    SELECT Capital AS Cap_City, State_Province AS St_Pr, Country AS Cntry
    FROM T_Capital_Cities
) AS Capitals
INNER JOIN
(
    SELECT Capital AS Cap_City, Cal_Year AS C_Year
    , Sum(Quart_Rainfall) AS Year_Rainfall
    FROM T_Capital_Rainfall_Q
    GROUP BY Capital, Cal_Year
) AS Yearly_Rainfall
ON Capitals.Cap_City = Yearly_Rainfall.Cap_City
ORDER BY Capitals.Cap_City;
```

Make sure what you see in the “Query pane” is exactly the code above, and if not, edit until it is exactly the same.

Right-click on the **tab** of the “Query pane” and in the pop-up menu click on “**Close**”. MS-Access will ask you if you want to save the changes in the Query, and you should click “**Yes**”. Then MS-Access will ask for the Query name: type-in “A\_Capitals\_data”

<sup>8</sup> The “**Esc**” key is also called the “**Escape**” key.

<sup>9</sup> This is the Query “A\_Capitals\_data” from file “Company\_Database.accdb”.

(always **without** quotes).

Notice this Query has **two Select** operations. The first one produces a record-list with all the capital data from Table “T\_Capital\_Cities”. The second **Select** operation produces a record-list with all the capital cities, a calendar year and the total yearly rainfall of that year. Now, the “**INNER JOIN**” SQL operator joins the records from each record-list, to produce new records with **combined fields**. The fields of the output record-list are the fields after the first “**SELECT**” clause of the Query. What records are **joined** is stated in the “**ON**” clause, which in this case states to **join** the records from both Tables where the value of the field “**Capital**” is the same.

Notice that in the SQL code above the “**AS**” clauses determine the **name** of the **output** columns. If you do not use an “**AS**” clause, and the “**SELECT**” **expression** consists of just **a field name**, then **that field name** is the **name** of the **output** column.

If you want a full description of a **Join** Query, you may click “[F.8 What is a Join operation and how do I write it?](#)”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

### **A.10.2 How do I run my first Join Query?**

If you just read [A.10.1](#), you should have an opened database with **two** Tables and a **Join** Query. Otherwise, click [A.10.1](#) and follow my indications until you reach here.

Double-click on “A\_Capitals\_data” placed in the “**Navigation Pane**” (click [B.4](#)). MS-Access will now open a “Query pane” in “**Datasheet View**”, with a **tab** at its top named “A\_Capitals\_data” showing the following record-list:

A_Capitals_data				
Cap_City	St_Pr	Cntry	C_Year	Year_Rainfall
Beijing		China	2018	28.8
Washington	District of Columbia	United States	2018	32.76

Check this output record-list against my explanation above and the SQL code, to see the effect of each SQL clause and operator. Notice that you will **not** see the colored fonts: I have added the colors above to help you relate the Query code with the Query output.

Once you are done, close this “Query pane” by right-clicking on its **tab** and in the pop-up menu click on “**Close**”.

### **A.11 How do I write and run my first Transform Query?**


You may click:

- “[A.11.1 How do I write my first Transform Query?](#)”
- “[A.11.2 How do I run my first Transform Query?](#)”

#### **A.11.1 How do I write my first Transform Query?**

If you just read [A.3](#) and [A.5](#), you should have an opened database with two configured Tables. Otherwise, click [A.3](#) and [A.5](#), follow my indications there, and then return here

(you return by simultaneously pressing the “Alt” and “←” keys).

Click on “Create” (placed right below the top window frame). This will cause the “Create” Ribbon (toolbar) to be shown. Now click on the **Query Design**  icon inside the “Create” Ribbon. This will open a “Query pane” called “Query5” or something similar. The “Query pane” has a **tab** at its top with the Query name “Query5”.

In addition to having created a “Query pane”, MS-Access has opened a box with a Table list and buttons “Add” and “Close”. Click on its “Close” button. Now, right-click on the **tab** at the top of the “Query pane” and in the pop-up menu click on “SQL View”. The “Query pane” is changed now to “SQL View”, which is all blank, with the text “SELECT;” in its top left corner. You should click inside the “Query pane” and type-in (or copy/paste) the following SQL code<sup>10</sup>:

```
TRANSFORM Avg(Quart_Rainfall) AS GenVals
SELECT Capital AS Cap_City, Cal_Year AS C_Year
, Sum(Quart_Rainfall) AS Yearly_Rainfall
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year
ORDER BY Capital
PIVOT Quart ;
```

Make sure what you see in the “Query pane” is exactly the code above, and if not, edit until it is exactly the same.

Right-click on the **tab** at the top of the “Query pane” and in the pop-up menu click “Close”. MS-Access will ask you if you want to save the changes in the Query, and you should click “Yes”. Then MS-Access will ask for the Query name: type-in “A\_Rainfall\_by\_Quarters”.

This Query will have as **output** fields the three **fields** that you can see in its “SELECT” clause, and **in addition**, it will **generate additional** “PIVOT” **fields** from the **distinct values** of the “PIVOT” **expression** (in this case the field name “Quart”). The **values** of the “PIVOT” **fields** are the **results** of the “TRANSFORM” **expression**, aggregated over the “GROUP BY” **expressions jointly with** the “PIVOT” **expression**. It is much simpler than it sounds, so the best is that you look at the output, listed in the next subsection (a few lines below).

For your reference, this type of Query is called a **Transform** Query, or a **crosstab** Query.

Notice that in the SQL code above I am using an **irrelevant** “AS” clause to assign a name to the “TRANSFORM” **expression**. I do this to mark in **blue** color the “TRANSFORM” **expression**, in order to relate it to the **output** of the **Transform** operation in the next section.

If you want a full description of a **Transform** operation, you may click “[F.10 What is a Transform operation and how do I write it?](#)”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

---

<sup>10</sup> This is the Query “A\_Rainfall\_by\_Quarters” from file “Company\_Database.accdb”.

### A.11.2 How do I run my first Transform Query?

If you just read A.11.1, you should have an opened database with **two** Tables and a **Join** Query. Otherwise, click A.11.1 and follow my indications until you reach here.

Double-click on “A\_Rainfall\_by\_Quarters” from the “**Navigation Pane**” (click B.4). MS-Access will now open a “Query pane” in “**Datasheet View**”, with a tab at its top named “A\_Rainfall\_by\_Quarters”. You should get the following record-list:

A_Rainfall_by_Quarters						
Capital	Cal_Year	Yearly_Rainfall	Q1	Q2	Q3	Q4
Beijing	2018	28,8	0	4	7,8	17
Washington	2018	32,76	12.13	5.67	2.26	12.7

Check this output record-list against my explanation above and the SQL code, to see the effect of each SQL clause and operator. Notice that you will **not** see the colored fonts: I have added the colors above to help you relate the Query code with the Query output.

Once you are done, close this “Query pane” by right-clicking on its **tab** and in the pop-up menu click on “**Close**”.

## PART B. BRIEFING ON MS-ACCESS USER INTERFACE




This **Part B** contains a description of the MS-Access user interface. I will focus on explaining the different elements, objects and views **that you can see** in MS-Access, indicating **many actions/commands** that you can use.

You may click:

- “B.1 What options should I set in MS-Access?”
- “B.2 How is the MS-Access window structured?”
- “B.3 What are the Ribbons?”
- “B.4 What is the “Navigation Pane” and how it works?”
- “B.5 What is a Table/Query/Form in “Datasheet View”?”
- “B.6 What is a Table in “Design View”?”
- “B.7 What is a Query in “Design View”?”
- “B.8 What is a Form in “Design View”?”
- “B.9 What is a Query in “SQL View”?”
- “B.10 What is the “Relationships” pane?”
- “B.11 Can I use a drop-down/expression menu even if its icon is not shown?”

### B.1 What options should I set in MS-Access?

It is convenient to begin by configuring the options from MS-Access in order to better adapt it to what you want. MS-Access allows to configure a number of options for the user interface, default values, preferred actions and so on.

Aside from the options you may set, I advise you make the Ribbon Area permanent. Click on the “**Home**” Ribbon name, placed on the top left corner of the MS-Access window. This will make the **Home** Ribbon (a “Ribbon” is a toolbar placed towards the top of the MS-Access window) appear. Click on the **pin** “” icon placed at the bottom right corner of the Ribbon. The **pin** “” icon will change to the **menu** “” icon and the **Ribbon area** will be shown **permanently** instead of being automatically hidden. A **permanent Ribbon area** will be much more convenient for you until you are more familiar with MS-Access.

To configure MS-Access options first click on “**File**” at the rightmost side of the “Ribbon-bar” (at the window top), and then click on “**Options**”. I recommend configuring the following options, because on my experience, **they adapt well** to the most frequent use of MS-Access. Options are very personal, so you may of course disagree with recommendations and set different ones.

If you want my advice on what options to set, you may click:

- “B.1.1 What “Current Database” options should I set?”
- “B.1.2 What “Object Designers” options should I set?”
- “B.1.3 What “Client Settings” options should I set?”

- “B.1.4 What “Quick Access Toolbar” options should I set?”

### B.1.1 What “Current Database” options should I set?

These options apply to **each database file**, so you will have to configure them for each new database file that you create.

Click the sequence: **File**→**Options**→**Current Database**.

Under the heading “**Application Options**”:

- **Tick** the checkbox for “**Tabbed Documents**”, under “**Document Window Option**”. This will allow you to open several objects at the same time, having each of them in a **tabbed** window. Otherwise, you will have overlapping windows which is far less convenient. Make sure the checkbox “**Display Document Tabs**” is **ticked**.
- **Untick** the checkbox for “**Enable design changes for tables in Datasheet view**”. This will prevent MS-Access from showing the column “**Click to Add**” in your Tables. I consider this column very **distracting**, and also, I think it is somehow **dangerous**, because the database user may unwillingly add a field to the Table by mistake.

Under the heading “**Navigation**”:

- **Click** on the button “**Navigation Options**”. In the dialog box that pops-up, **tick** the checkbox “**Single-click**”, that is placed under “**Open objects with**”. You can also open the “**Navigation Options**” dialog box with a right-click anywhere in the “**Navigation Pane**” heading and clicking on “**Navigation Options**” from the pop-up menu.

By **ticking** the “**Single-click**” option you will be able to **select** objects (Tables, linked Tables, Queries, Forms, Reports and Modules) in the “**Navigation Pane**” (click just by **placing** the mouse-pointer **on top** of the object name. It will also allow you to **open** the objects with just **one mouse click** instead of with a double-click. This is very convenient, and I advise you set this option.

**Be aware** that setting “**Single-click**” causes a **somehow annoying side effect** when **renaming** objects. When you want to rename an object, you right-click on the object’s name, and then click on “**Rename**” from the pop-up menu. This highlights the object **name** and you can edit it with the keyboard, pressing the “**Enter**” key when you are done. **However**, if the “**Single-click**” option is set, **and** the mouse stays within the “**Navigation Pane**” after you clicked on “**Rename**”, the renaming **will not work**. The reason is that “**Single-click**” implies that **objects** are **selected** by just placing the mouse-pointer over them: therefore, if the mouse-pointer stays (after having clicked on “**Rename**”) **within** the “**Navigation Pane**”, the **object name** below the mouse-pointer will be **selected**, and the object name that had just been selected for renaming will be **cleared**. The solution is really simple (just **remind** about it): when you want to rename an object, after having clicked on “**Rename**”, you **must** immediately move the mouse-pointer out of the “**Navigation Pane**”. Once the mouse-pointer is out of the navigation pane, you may edit the object name with the keyboard as described above.

### B.1.2 What “Object Designers” options should I set?

These options apply to **each database file**, so you will have to configure them for each new database file that you create.

Click the sequence **File**→**Options**→**Object Designers**.

Under the heading “**Table design view**”:

- Set **Default field type=Number**  
When you create a new field in a Table, the default type of the field will be “**Number**”.
- Set **Default text field size=255**  
The default field size for a **Short Text** field will be 255 characters, which is the maximum one.
- Set **Default number field size=Double**  
When you create a new field in a Table, the default “**Size**” property for the field type “**Number**” will be “**Double**”. This means that the field is a double precision floating-point data type.

### B.1.3 What “Client Settings” options should I set?

These options apply to **each MS-Access** installed program (i.e., in each laptop or Windows account that you use), so you will have to configure them for each different MS-Access installed program that you use.

Click the sequence **File**→**Options**→**Client Settings**.




Under the heading “**Editing**”:

- Set **Default find/replace behavior=General Search**  
This will cause that the default behavior of the “**Find/Replace**” tool will be to search **in any field** of the Query/Table you are searching in, and also, to match any **substring** of the field value (i.e., do not require an exact match). On my experience, this is **by far** the behavior you want on most occasions, so I **strongly** recommend you set this option.

### B.1.4 What “Quick Access Toolbar” options should I set?


These options apply to **each MS-Access** installed program (i.e., in each laptop or Windows account that you use), so you will have to configure them for each different MS-Access installed program that you use.

Click the sequence **File**→**Options**→**Quick Access Toolbar**

- Add the **Find** tool to the “**Quick Access Toolbar**”  
This will put the shortcut **Find** “” icon in the “**Quick Access Toolbar**” (click *B.2.2*). On my experience, the “**Find**” tool (click *B.5.5*) is used **very** frequently, so it is very useful to have it in the “**Quick Access Toolbar**”.
- Add the **Undo** and **Redo** tools to the “**Quick Access Toolbar**”  
This will put the shortcut **Undo** “” and **Redo** “” icons in the “**Quick Access Toolbar**” (click *B.2.2*). On my experience, these tools are used somehow frequently,



so it is useful to have them in the “**Quick Access Toolbar**”.

- In case you have purchased the plug-in “**Access SQL Editor**” (click *F.5*), you should **definitively** add its tool “” icon to the “**Quick Access Toolbar**” (click *B.2.2*), because you will use it **a lot**.
- You should also add other commands that **you** want to use **frequently** to the “**Quick Access Toolbar**”.

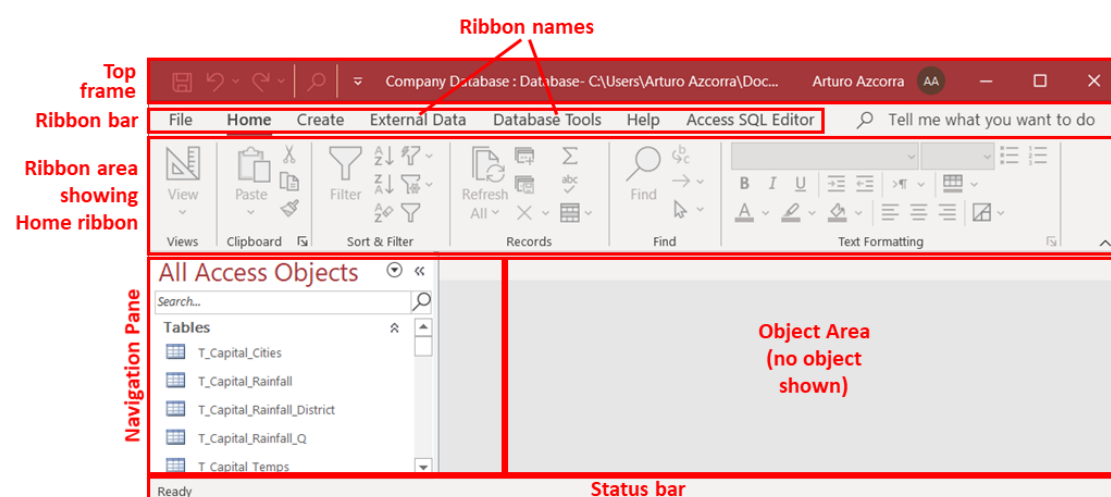
Notice that some of these options only take effect after closing MS-Access and launching it again, so you should do this after you have changed your options. Notice also that some options affect the behavior of MS-Access itself, while others are specific of the database file where you have applied them.

## B.2 How is the MS-Access window structured?

When you have a **database file open**, you will see the following relevant elements in the MS-Access window, starting from the top and going downwards:

- **Top window frame** (wide dark red bar at the window top). It includes inside it the “**Quick Access Toolbar**” (at the left, composed of tool icons), the file name and other elements.
- “**Ribbon-bar**” (list of Ribbon names, placed right below the top window frame).
- **The currently selected Ribbon (if not hidden, it is below the top window frame)**.
- “**Navigation Pane**” (below the Ribbon, on the left side).
- **Object Area** (below the Ribbon, on the right side).
- MS-Access “**Status-bar**” (at the very bottom of the window, below both the “**Navigation Pane**” and the “**Object Area**”).

The following screenshot shows the elements listed above:



You may also click on the following sections:




- “*B.2.1 What is the top window frame?*”
- “*B.2.2 What is the “Quick Access Toolbar”?*”



- “B.2.3 What is the “Ribbon-bar”?”
- “B.2.4 What are Ribbon names?”
- “B.2.5 What is the Ribbon Area?”
- “B.2.6 What is the visible Ribbon?”
- “B.2.7 What is the “Navigation Pane”?”
- “B.2.8 What is the Object Area?”
- “B.2.9 What are the object tabs?”
- “B.2.10 What is the Visible Object and the object panes?”
- “B.2.11 What is the “Status-bar”?”

### **B.2.1 What is the top window frame?**

The top MS-Access window frame is a thick bar in dark red at the very top of the MS-Access window and it contains, left to right:

- The “**Quick Access Toolbar**” (list of tool icons).
- Depending on context: “Contextual Ribbon label” (click *B.3.4.2*).
- The name of the currently open database file.
- The folder path to the currently open database.
- The three usual **window icons** placed at the **topmost** and **rightmost** side of any window: the **minimize window** “” icon, the **maximize window** “” icon, and the **close window** “” icon.

The “Contextual Ribbon label” may be empty or a label may be shown. Whether it is empty, and what label is shown (if any) depends on **what is the object currently viewed** in the “**Object Area**”. If you want to know more about contextual Ribbons, you may click “*B.3.4.1 What are contextual Ribbons?*”.

### **B.2.2 What is the “Quick Access Toolbar”?**

It is placed inside the top frame of the MS-Access windows, in its leftmost side. It contains icons of commands and tools to do different actions on your database. You can configure what commands are placed in the “**Quick Access Toolbar**”. To do it, click on “**File**”, then on “**Options**” and then on “**Quick Access Toolbar**”. There you can add or remove icons and/or change their order.

### **B.2.3 What is the “Ribbon-bar”?**

It is placed right below the top window frame. It has no icons, and it is composed of a list of **Ribbon names**. **Clicking** on any of the Ribbon names (except “**File**”) will **display** the corresponding Ribbon in the Ribbon Area (click *B.2.5*). Notice you can **only see one Ribbon at a time** in the Ribbon Area. The **name of the currently visible Ribbon** will be highlighted and **underlined** in the “Ribbon-bar”.

### **B.2.4 What are Ribbon names?**

The Ribbon names are placed **inside** the “Ribbon-bar”. A **Ribbon name** is the name of

each Ribbon. In Microsoft's terminology "Ribbon names" are called "Ribbon-bar tabs", but I will not use this term to avoid confusion with the Table tabs and Query tabs.




**Clicking** on any of the Ribbon names (except "File") will **display** the corresponding Ribbon in the Ribbon Area. Notice you can **only** see **one Ribbon at a time**. The **name of the currently visible Ribbon** will be highlighted and **underlined** in the "Ribbon-bar".




Notice that some Ribbon names **are only shown depending on** what **object** is currently **visible** in the "Object Area". These Ribbon names that are only shown depending on what object is shown in the Ribbon Area are called "**contextual Ribbon names**" (click *B.3.4.3*). The Ribbons corresponding to contextual Ribbon names are called "**contextual Ribbons**" (click *B.3.4.1*). If you want to know more about **contextual Ribbons**, you may click "*B.3.4 What contextual Ribbon can I display?*".

### **B.2.5 What is the Ribbon Area?**

It is the area below the "Ribbon-bar", where the **visible Ribbon** is shown. A **Ribbon** is a wide horizontal toolbar containing icons and commands grouped in **Ribbon groups**.

There are two configurations for the Ribbon Area: **permanent** or **pop-up**. In the **permanent configuration**, the Ribbon Area is **always visible**. In the **pop-up configuration**, the Ribbon Area is **only visible** after having **clicked** on a Ribbon name, and it will be hidden when you click on some other part of the MS-Window.

To configure the **Ribbon Area as permanent**, click on any Ribbon name (except "File") to show it, and then click on the **pin** " icon, placed in the bottom-right corner of the Ribbon Area. This will configure the Ribbon Area as permanent, and the **pin** " icon will change to the **menu** " icon.

To configure the **Ribbon Area as pop-up**, click on the **menu** " icon placed in the bottom-right corner of the Ribbon Area. This will hide the Ribbon Area, configure the Ribbon Area as pop-up, and the **menu** " icon will change to the **pin** " icon.


### **B.2.6 What is the visible Ribbon?**


The currently **visible** Ribbon is placed in the Ribbon Area (click *B.2.5*), right below the "Ribbon-bar". Each Ribbon is a **toolbar** with **many** commands, icons and tools. There are **many** Ribbons, however, you can **only view one Ribbon at a time**. To **view a Ribbon**, you only need to **click** on the corresponding Ribbon **name**, placed in the "Ribbon-bar" (click *B.2.3*). Notice that some Ribbon names (the **contextual Ribbon names**) are not shown permanently, so therefore, you can only select the corresponding Ribbons when the Ribbon name is shown. If you want to know more about this, you may click "*B.3.4 What contextual Ribbon can I display?*".

The **name of the currently visible Ribbon** will be highlighted and **underlined** in the "Ribbon-bar".

### **B.2.7 What is the "Navigation Pane"?**

The "**Navigation Pane**" is placed below the Ribbon Area, at the left side of the MS-Access window. It **lists** the **names** of the objects (Tables, Queries, ...) that exist in the database. If you double-click (do two quick clicks) on any object, it will be opened in the "Object Area".

You can **hide** the “**Navigation Pane**” by clicking on the **hide** “” icon placed in its top-right corner. If you hide it, a **vertical** bar labeled “**Navigation Pane**” will remain at the leftmost part of the MS-Access window.

You can **display** the “**Navigation Pane**” by clicking on the **unhide** “” icon at the top of the **vertical** bar placed at the left of the window, labeled “**Navigation Pane**”.

If you want to know more about the “**Navigation Pane**” you may click “*B.4 What is the “Navigation Pane” and how it works?*”.

## **B.2.8 What is the Object Area?**







It is the MS-Access window area placed below the “**Ribbon Area**” and to the right of the “**Navigation Pane**”. The “**Object Area**” can contain zero, one, or more **opened objects**. If it contains **zero opened objects**, the “**Object Area**” is **completely empty**. If it contains one or more **opened objects**, the top part of the “**Object Area**” will show **one “Object tab”** (click *B.2.9*) **for each opened object**. The rest of the “**Object Area**” will show **one and only one** of the **opened “Objects”**, called the “**Visible Object**” (click *B.2.10*).

You can select which **object** is visible in the “**Object Area**” by clicking on its “**Object tab**” at the top of the “**Object Area**” (click *B.4.1.5*).

## **B.2.9 What are the object tabs?**

An **object tab** is the **tab** placed at the top of each **object pane** (click *B.2.10*).

Each “**object tab**” contains an **icon** (that identifies the object **type**) placed on its left side, the object **name** to the right of the icon, and a close “**X**” icon on its rightmost side. There can be many “**Object tabs**”, but **only one object is visible** at the “**Object Area**” at any given moment.

If the MS-Access window width is **too narrow** to show **all** the tabs of opened objects, MS-Access will show a **next left** “” icon on the **leftmost side** and/or a **next right** “” icon on the **rightmost side** of the **object tabs**. The **next left** “” icon is shown when there are **hidden tabs** to the **left** of the visible ones. The **next right** “” icon is shown when there are **hidden tabs** to the **right** of the visible ones. If you click on the **next left** “” icon, the **first hidden** tab on the **left** side will become visible, and the required ones (depending on their width) on the **right** side will become hidden. If you click on the **next right** “” icon, the **first hidden** tab on the **right** side will become visible, and the required ones (depending on their width) on the **left** side will become hidden.

You can select which **object** is visible in the “**Object Area**” by clicking on its “**Object tab**” at the top of the “**Object Area**”.

By default, “**Object tabs**” are **ordered** left to right by the moment when the object was opened: object opened **longest ago is leftmost**, and object opened **most recently is rightmost**. However, you can change the order of **tabs** by doing drag-and-drop over the tab that you want to move.

## **B.2.10 What is the Visible Object and the object panes?**

The **Visible Object** is **the object**, among all currently **opened object**, that is **visible** in the “**Object Area**”. To select which among the opened objects is the visible one, click

on its **object tab**, at the top of the “Object Area”.

The “Visible Object” is shown in an object pane, below the “Object tabs”. The structure and elements of **each** object pane **depend** on the **type of object** (e.g., Table, Query, Form, ...) and on the **view-type** (e.g., “**Datasheet View**”, “**Design View**”, “**SQL View**”, ...) of the object. If you want to know more about the object panes, you may click:

- “B.5 *What is a Table/Query/Form in “Datasheet View”?*”
- “B.6 *What is a Table in “Design View”?*”
- “B.7 *What is a Query in “Design View”?*”
- “B.8 *What is a Form in “Design View”?*”
- “B.9 *What is a Query in “SQL View”?*”
- “B.10 *What is the “Relationships” pane?*”

The Visible Object determines what “Contextual Ribbon names” (click B.3.4.3) you will see in the “Ribbon-bar”, as I indicated when explaining the Ribbons above.

### **B.2.11 What is the “Status-bar”?**

The “**Status-bar**” is placed at the very bottom of the MS-Access window, right above of the bottom window frame. The “Status-bar” presents some information on the objects and commands you are working with, and also, a few contextual buttons. When contextual buttons are shown, they are placed in the rightmost side of the “Status-bar”.

One example of contextual buttons are the ones that allow you to change the **view-type** (“**Datasheet**”, “**Design**”, “**Layout**”, ...) of the object that you are currently viewing in the “Object Area” (click B.4.1.4).

## **B.3 What are the Ribbons?**

A **Ribbon** is a wide horizontal toolbar containing icons and commands grouped in **Ribbon groups**.

You may click:

- “B.3.1 *What is a Ribbon group?*”
- “B.3.2 *Why do Ribbons change how they look?*”
- “B.3.3 *What permanent Ribbon can I display?*”
- “B.3.4 *What contextual Ribbon can I display?*”

### **B.3.1 What is a Ribbon group?**

It is a group of tool icons (**within** a Ribbon) that perform related functions. A Ribbon is divided in **Ribbon groups**. Each **Ribbon group** is delimited by a vertical line in the Ribbon. Each Ribbon group is identified by a Ribbon group **name** shown at the bottom part of the Ribbon. The following screenshot shows the “**Home**” Ribbon, with its six


Ribbon groups, each with its Ribbon group name at the bottom.



In this example, the six **Ribbon group** names are “**Views**”, “**Clipboard**”, “**Sort & Filter**”, “**Records**”, “**Find**” and “**Text Formatting**”. Some Ribbon groups have a “Dialog box launcher” icon in their bottom-right corner (see it in the screenshot above). Clicking on the “Dialog box launcher” will show a dialog box where more settings and configuration options are shown, all related to the Ribbon group. In the screenshot above you may see that the Ribbon groups “**Clipboard**” and “**Text Formatting**” have a “Dialog box launcher”, while the other Ribbon groups do not have it.

### B.3.2 Why do Ribbons change how they look?

Ribbons are a little **puzzling** initially because they are **not always the same**.

First, the Ribbon Area is sometimes **hidden** and sometimes **shown**. You may make the Ribbon Area **permanently shown** by clicking on the pin “” icon located in its bottom right corner (click *B.1*).

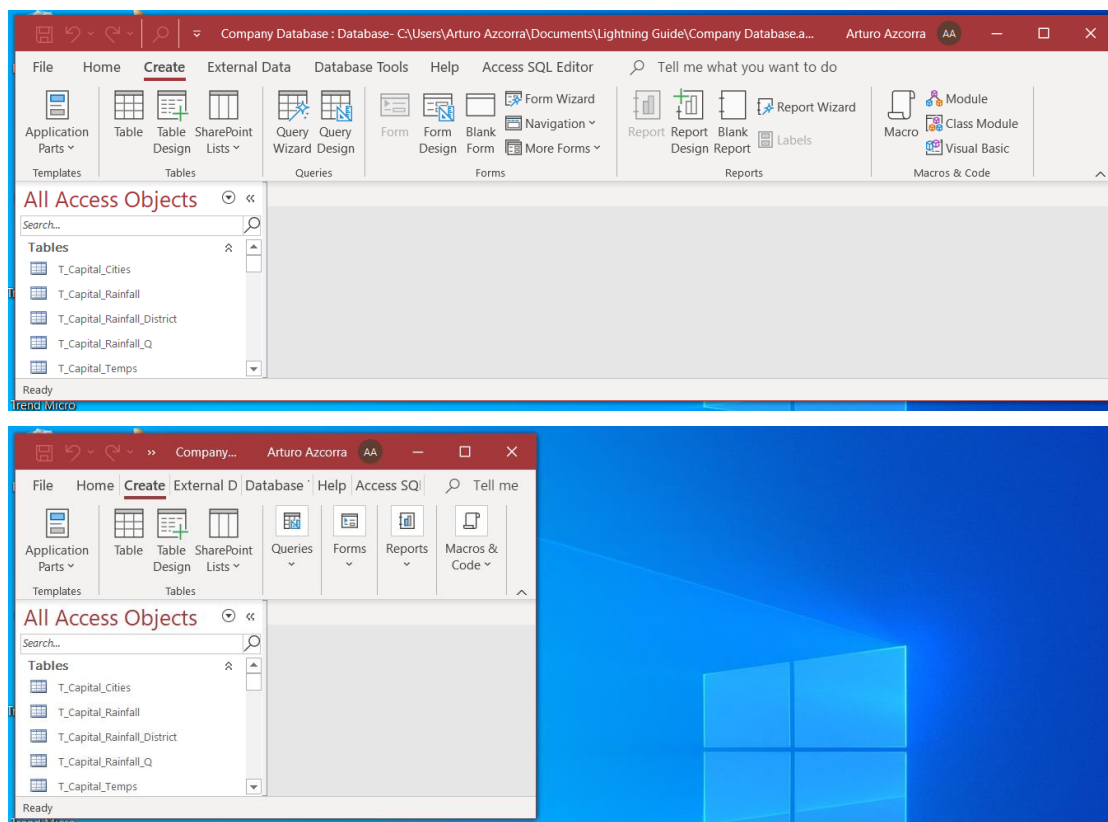
Second, the Ribbon that is shown in the Ribbon Area **changes** each time you click on a Ribbon name (click *B.2.4*).

Third, some **Ribbon names** are **not always shown**. This is because some Ribbon names (called “**contextual Ribbons**”: click *B.3.4*) are only shown if a given **object** (Table in “**Design View**”, Table in “**Datasheet View**”, Query, ...) is **visible** in the “Object Area”.

Fourth, some **contextual Ribbon names** are **duplicated**, and you have to take **also** into account what **contextual Ribbon label** is shown to identify which they are.

Finally, the **command icons** in each Ribbon are **changed** and **relocated** depending on the MS-Access window **width**. When you make the window narrower, and all the icons do not fit, some icon name-tags will be hidden. If you make the window even narrower, **some** icons will be **grouped** and **replaced** by a drop-down menu icon: clicking on the icon will show a pop-up menu with its tool icons. If you make the window narrower still, a **complete** Ribbon group is **replaced** by a drop-down menu icon: clicking on the icon will show all the tools in the Ribbon group.

Check the following two screenshots and notice how the “**Create**” Ribbon changes very significantly when making the MS-Access window smaller:



As you may see, the Ribbon groups “**Queries**”, “**Forms**”, “**Reports**” and “**Macros & Code**” have been each **squeezed** into just one drop-down menu. If you click on one of these drop-down menus you will find **all** the tools of the corresponding Ribbon group.


### B.3.3 What permanent Ribbon can I display?

You can display (becoming the visible Ribbon) one of the permanent Ribbons by clicking on the corresponding Ribbon name. The list of Ribbon names is the following (as can be found left to right in the “Ribbon-bar”):

- “**Home**” Ribbon  
Contains **general purpose** commands, like **cut**, **paste**, **search** or “**Refresh all**”.
- “**Create**” Ribbon  
Contains commands to **create** new database **objects**: Tables, Queries, Forms, Reports and Macros.
- “**External Data**” Ribbon  
Contains commands to **import**, **export** and do other operations **with external data**.
- “**Database Tools**” Ribbon  
Contains commands to **view and create Relationships**, **compact and repair the database**, **run macros**, **present object dependencies**, and other database operations.



- **“Plug-in” Ribbon**

In case you incorporate a plug-in tool for MS-Access it may add its own Ribbon. This is the case of the plug-in **“Access SQL Editor”** (the one I recommend you to buy). If you install it, it will add the Ribbon name **“Access SQL Editor”** to the “Ribbon-bar”. If you click on that Ribbon name, you will see a large **Access SQL Editor**  icon.

- **Depending on context: contextual Ribbon**

See the next section *B.3.4* right below for an explanation.

### **B.3.4 What contextual Ribbon can I display?**

You may click:

- *“B.3.4.1 What are contextual Ribbons?”*
- *“B.3.4.2 What are contextual Ribbon labels?”*
- *“B.3.4.3 What are contextual Ribbon names?”*
- *“B.3.4.4 What contextual Ribbon can I display, depending on context?”*

#### **B.3.4.1 What are contextual Ribbons?**

**Contextual Ribbons** are Ribbons that **you can only select** when its **Ribbon label** and its **Ribbon name** are shown. A contextual Ribbon is shown in the “Ribbon Area” as any other Ribbon.

#### **B.3.4.2 What are contextual Ribbon labels?**

**Contextual Ribbon labels** are placed (when shown) in the top window frame. A contextual Ribbon **label** is a text string placed on top of the contextual Ribbon names, that indicates what type of contextual Ribbons are currently available. **Contextual Ribbon labels** are **not shown all the time**.

#### **B.3.4.3 What are contextual Ribbon names?**

**Contextual Ribbon names** are Ribbon names that are **not shown all the time**. The **contextual Ribbon names** are placed (when shown) **in the “Ribbon-bar”**, at the right of the permanently shown Ribbon names.

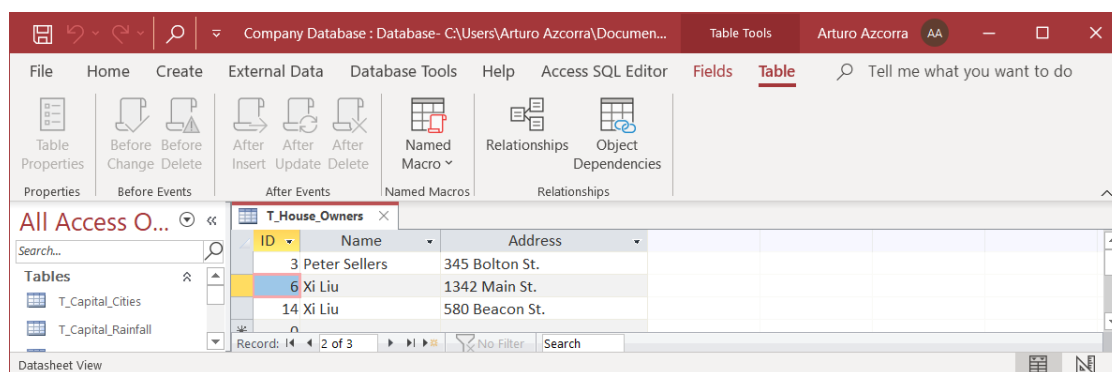
#### **B.3.4.4 What contextual Ribbon can I display, depending on context?**

You can display (becoming the visible Ribbon) one of the contextual Ribbons that are available depending on context.

The **contextual Ribbon** that you can **display** at any given moment depends **both** on the **contextual Ribbon label shown**, and also, on the **contextual Ribbon name shown**. You need **both** because some **contextual Ribbon names** are **duplicated**, and you need to know **also** the **contextual Ribbon label** to identify the contextual Ribbon. For example, the contextual Ribbon **name** **“Design”** under the Ribbon **label** **“Table Tools”** will open a different Ribbon than the **same** contextual Ribbon name **“Design”** under the Ribbon **label** **“Query tools”**.

The next screenshot shows the contextual Ribbon label **“Table Tools”** and the contextual Ribbon names **“Fields”** and **“Table”** under it. The Ribbon shown in the Ribbon area is

“**Table**” (you may see that its name is underlined):



The **contextual Ribbon label** and **contextual Ribbon names** that are **shown** at any given moment depend on **what object is shown** in the “**Object Area**”. The following list indicates what **contextual Ribbon label** and what **contextual Ribbon names** will be shown depending on the **type of object** opened, and on the **view-type** in which the object is opened in the “**Object Area**”:

- **No object in the “Object Area” => no contextual Ribbon label**  
No contextual Ribbon.
- **“Table pane” => “Table Tools” as contextual Ribbon label**
  - **“Datasheet View”**  
“**Fields**” and “**Table**” as contextual Ribbon **names** from the “Ribbon-bar”, both placed right below the contextual Ribbon label.
  - **“Design View”**  
“**Design**” as contextual Ribbon **name** from the “Ribbon-bar”, placed right below the contextual Ribbon label.
- **“Query pane”**
  - **“Datasheet View” => no contextual Ribbon label**  
No contextual Ribbon.
  - **“Design View” or “SQL View” => “Query Tools” as contextual Ribbon label**  
“**Design**” as contextual Ribbon **name** from the “Ribbon-bar”, placed right below the contextual Ribbon label.
- **“Form pane”**
  - **“Form View” => no contextual Ribbon label**  
No contextual Ribbon.
  - **“Datasheet View” => “Form Tools” as contextual Ribbon label**  
“**Datasheet**” as contextual Ribbon **name** from the “Ribbon-bar”, placed right below the contextual Ribbon label
  - **“Layout View” => “Form Layout Tools” as contextual Ribbon label**  
“**Design**”, “**Arrange**” and “**Format**” as contextual Ribbon **names** from the “Ribbon-bar”, placed right below the contextual Ribbon label.
  - **“Design View” => “Form Design Tools” as contextual Ribbon label**  
“**Design**”, “**Arrange**” and “**Format**” as contextual Ribbon **names** from the “Ribbon-bar”, the three placed right below the contextual Ribbon label.



- **“Report pane”**
  - **“Report View”** => no contextual Ribbon label  
No contextual Ribbon.
  - **“Layout View”** => **“Report Layout Tools”** as contextual Ribbon **label**  
**“Design”**, **“Arrange”**, **“Format”** and **“Page Setup”** as contextual Ribbon **names**  
from the “Ribbon-bar”, the four placed right below the contextual Ribbon label.
  - **“Design View”** => **“Report Design Tools”** as contextual Ribbon **label**  
**“Design”**, **“Arrange”**, **“Format”** and **“Page Setup”** as contextual Ribbon **names**  
from the “Ribbon-bar”, the four placed right below the contextual Ribbon label.
  - **“Print Preview”**  
**All other normal Ribbon names are hidden** except **“File”**, and also, you will see **“Print Preview”** as contextual Ribbon name from the “Ribbon-bar”, and **no** contextual Ribbon **label** is shown.
- **“Macro pane”** => **“Macro Tools”** as contextual Ribbon **label**  
**“Design”** as contextual Ribbon **name** from the “Ribbon-bar”, placed right below the Ribbon label.
- **“Relationships” pane** => **“Relationship Tools”** as contextual **label**  
**“Design”** as contextual Ribbon **name** from the “Ribbon-bar”, placed right below the Ribbon label.

## **B.4 What is the “Navigation Pane” and how it works?**

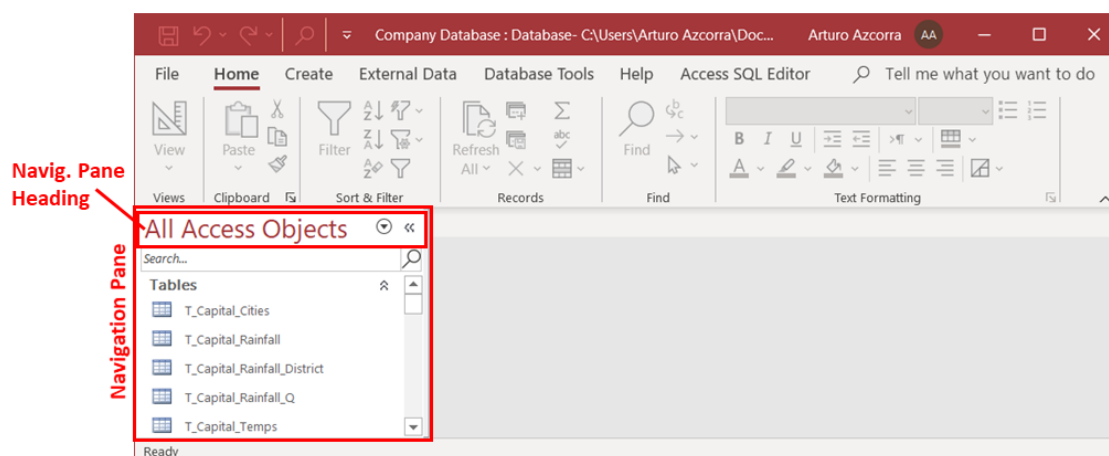
The **“Navigation Pane”** is placed below the Ribbon Area, at the left side of the MS-Access window, with the default title **“All Access Objects”** at its top. The **“Navigation Pane”** lists the **names** of all the objects that exist in the database file. Each database object belongs to one of the following object collections: **Tables**, **Queries**, **Forms**, **Reports** and **Modules**. **Object names within each object collection must be unique**, but object names can be **duplicated** if objects are of **different collections** (click D.2.5).

If you (double<sup>11</sup>) click on any object **name**, the corresponding object will be **opened** in the “Object Area” (except **Modules**, that are opened in the window of the VBA editor). You can also **open** an object directly in a specific **view-type**: right-click on the object **name** and in the pop-up menu click on the specific **view-type** you want. You can open objects in other ways, and also do other actions on them (click B.4.1).

---

<sup>11</sup> If you configure the option **“Single-click”** you will **open** an object with one click (instead of with a double-click), and also, you will **select** an object by just placing the mouse over it (instead of by clicking on it).


The next screenshot shows the “**Navigation Pane**”, on the left side of the MS-Access window, with the its heading (at the top)including the title “**All Access Objects**”:





The database object **names** are **grouped** in the “**Navigation Pane**” by **object name groups**. Each **object name group** has a specific **heading**, that is followed by a list of the object names in that group. If there are **no objects** in a given object name group, the corresponding **object name group heading** is **not** shown.

By default, the database object names are **grouped** by **object collection**, in the following top-down order: **Tables** (topmost), **Queries**, **Forms**, **Reports** and **Modules** (bottommost). You can **configure** how object names are **grouped** (click *B.4.2.1*).

By default, the database object names **within each group** are **ordered alphabetically** by object name. You can **configure** how object names are **ordered** (click *B.4.2.2*).

By default, **object name** are **displayed** as a **list**, with a small **icon** to the left of each object name that identifies its object **type**. These icons are: **Table** “

You can **hide** the “**Navigation Pane**” by clicking on the **hide** “

You can **unhide** the “**Navigation Pane**” by clicking on the **unhide** “

You can **change the width** of the “**Navigation Pane**” by doing click and drag on the vertical line that separates it from the “**Object Area**”.

If you want to know more about the “**Navigation Pane**”, you may click:

- “*B.4.1 How do I open, close and do other actions on objects in the “Navigation Pane”?*”
- “*B.4.2 How do I configure the way object names are grouped, ordered, displayed or hidden in the “Navigation Pane”?*”

### **B.4.1 How do I open, close and do other actions on objects in the “Navigation Pane”?**

This section applies to the objects from the “**Navigation Pane**”, which are **Tables**, **linked Tables**, **Queries**, **Forms**, **Reports** and **Modules**. **Relationships** are **not** objects from the “**Navigation Pane**”: if you want to handle Relationships, click “*B.10 What is the “Relationships” pane?*”.

Coming back to the objects from the “**Navigation Pane**”, you may click:

- “*B.4.1.1 How do I search for an object name in the “Navigation Pane”?*”
- “*B.4.1.2 What is the object menu in the “Navigation Pane”?*”
- “*B.4.1.3 How do I open an object with the “Navigation Pane”?*”
- “*B.4.1.4 How do I change the view-type of an opened object?*”
- “*B.4.1.5 How do I view another of the opened objects?*”
- “*B.4.1.6 How do I save the layout/design of an opened object?*”
- “*B.4.1.7 How do I close an opened object?*”
- “*B.4.1.8 How do I rename an object?*”
- “*B.4.1.9 How do I run a Query?*”
- “*B.4.1.10 How do I select one or more objects?*”
- “*B.4.1.11 How do I edit a Query?*”
- “*B.4.1.12 How do I create an object?*”
- “*B.4.1.13 How do I delete objects?*”
- “*B.4.1.14 How do I copy/cut and paste objects?*”

#### **B.4.1.1 How do I search for an object name in the “Navigation Pane”?**

The “**Navigation Pane**” has an object search bar at its top that allows you to search for object names by **substring**. If you **type-in** a **string** in the object search bar, *only* the objects whose **name** contains **that string** will be **shown**. This is done **as you type**, so the more characters you type, the fewer objects are shown.

You can hide/unhide the search bar in either of the following ways:

- Right-click anywhere in the “**Navigation Pane**” heading and click on “**Search Bar**” from the pop-up menu.
- Right-clicking anywhere in the “**Navigation Pane**” heading and click on “**Navigation Options**” from the pop-up menu. A sub-window will be shown and there you may untick/tick the checkbox “**Show Search Bar**”.

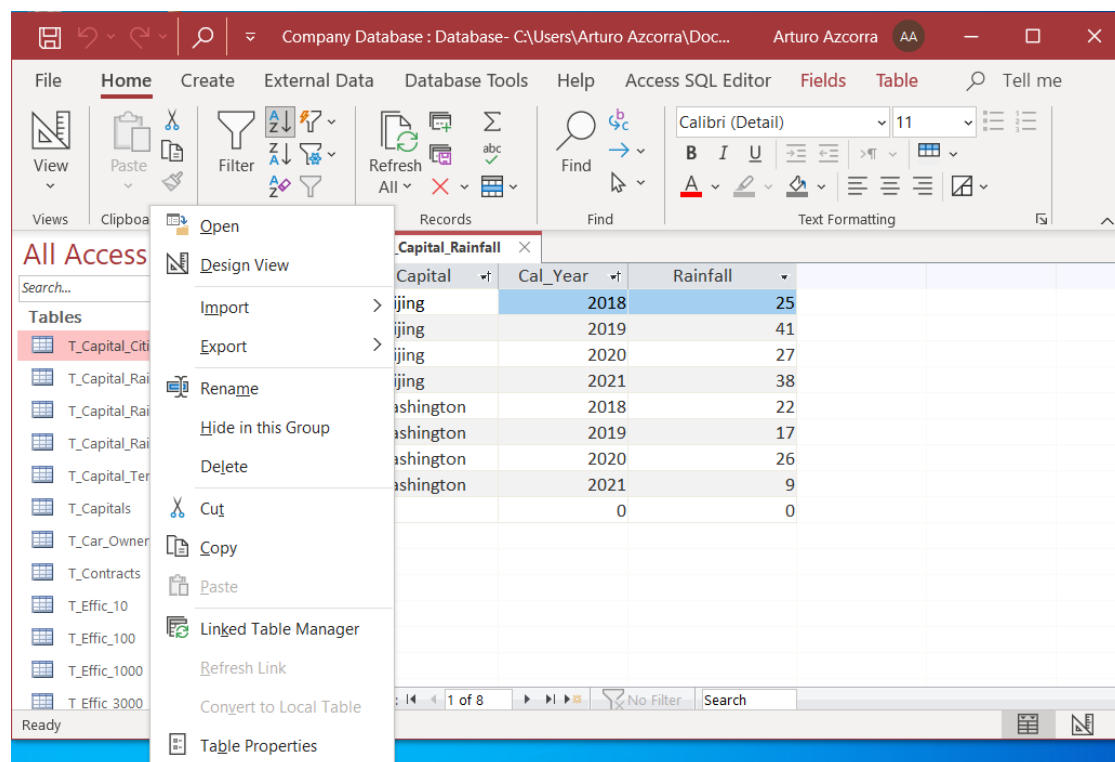
My advice is you always keep the object search bar shown because it is very useful.

#### **B.4.1.2 What is the object menu in the “Navigation Pane”?**

It is the **menu** with the **actions** that you can do **over objects** from the “**Navigation**

Pane”.

You show the **object menu** by right-clicking on an **object name** from the “**Navigation Pane**”. The next screenshot shows the **object menu** you get for a **Table**:



The **actions** in the **object menu** depend on the object type, as follows.

**Menu actions common to all the object types:**

- “**Open**”  
**Opens the Table, linked Table, Form or Report** in “**Datasheet View**” (click *B.4.1.3*).  
**Runs the Query** and presents its results in “**Datasheet View**” (click *B.4.1.9*).  
**Opens the VBA Module** in the specific window of the VBA editor (click *K.9.1*).
- “**Design View**”  
**Opens the Table, linked Table, Query, Form or Report** in “**Design View**” (click *B.4.1.3*).  
**Opens the VBA Module** in the specific window of the VBA editor (click *K.9.1*).
- “**Export**”  
**Exports the object** to one of the **formats** that are available in the **submenu** that is shown when you place the mouse over the “**Export**” row of the **object menu**.
- “**Rename**”  
**Allows you to change the object name** (click *B.4.1.8*).
- “**Hide in this Group**”  
**Hides the object** (click *B.4.2.5*).
- “**Delete**”  
**Deletes the object** (click *B.4.1.13*). Notice that deleting an object **cannot be**

undone.

- “**Cut**” or “**Copy**”  
**Cuts** or **copies** (respectively) the **object** (click *B.4.1.14*). Notice that you can **cut/copy** and **paste objects** (Tables, Queries, ...) between different MS-Access instance (windows), which is **extremely** useful.
- “**Paste**”  
**Pastes** the **object** previously **cut/copied** in either this MS-Access instance (window), or another MS-Access instance (window). Notice that you can **cut/copy** and **paste objects** (Tables, Queries, ...) between different MS-Access instances (windows), which is **extremely** useful. In case you had not previously copied an object, “**Paste**” will be shadowed (see the screenshot above) and clicking on it will have no effect.
- “**Table Properties**”, “**Object Properties**”, “**View Properties**”  
**Shows** the “**Navigation Pane**” **properties** of the **object**. These are the creation and modification dates, the object owner and a few other “**Navigation Pane**” **properties**. **Do not mistake** the “**Navigation Pane**” **properties** of any object with the “**Properties Sheet**” of Tables (click *B.6.3*), Queries (click *B.7.1*) and Forms (click *B.8.1*). The latter are **much more useful and important**.

Menu actions specific of Tables and linked Tables:

- “**Import**”  
**Imports** content into the **Table** or **linked Table** from one of the formats available in the **submenu** shown when you place the mouse over the “**Import**” row of the **object menu**.
- “**Linked Table Manager**”  
**Opens** the “**Linked Table Manager**” (click *K.3.10*).

Menu actions specific of linked Tables:

- “**Refresh Link**”  
**Refreshes** the **link** of the **linked Table** (click *K.3.7*).
- “**Convert to Local Table**”  
**Converts** the **linked Table** into a **local Table** (click *K.3.8*).

Menu actions specific of Forms and Reports:

- “**Layout View**”  
**Opens** the **Form** or **Report** in “**Layout View**”.

Menu actions specific of Reports:

- “**Print...**” and “**Print Preview**”  
**Prints** or **displays** a **print preview** (respectively) of the **Report**.

### B.4.1.3 How do I open an object with the “Navigation Pane”?

You can **open** an object in either of the following ways:

- (Double<sup>12</sup>) Click on the object **name** in the “**Navigation Pane**”.
- Right-click (i.e., click with the right mouse button) on the object **name** in the “**Navigation Pane**” and click on “**Open**”<sup>13</sup> from the pop-up menu.
- Select the object (click *B.4.1.10*) and then press the “**Enter**” key.
- Right-click (i.e., click with the right mouse button) on the object **name** in the “**Navigation Pane**” and in the pop-up menu click on the specific **view-type** (e.g., “**Design View**” or “**Layout View**”) that you want.

If you use either of the **three first** ways, a **Table**, **Table link**, **Query**, **Form** or **Report** is opened in “**Datasheet View**” in the “Object area”.




If you use the fourth way, the **Table**, **Query**, **Form** or **Report** will be opened in the **specific view-type** that you selected. For the case of a **Module**, it will be opened in the window of the VBA editor.

Notice that not all the view-types are shown in this pop-up menu. For example, the “**SQL View**” is not shown. If you want a Query in “**SQL View**” you have to open it first in a different **view-type**, and then **change its view-type** (click *B.4.1.4*) to “**SQL View**”. Notice also that for the case of **Union Queries**, if you click on the view-type “**Design View**”, they will be opened in “**SQL View**”, because they do not have a “**Design View**” (click *K.4.7*). Finally, if you open a Query in “**Design View**”, but its “**Design View**” has not been ever saved, it will be opened in “**SQL View**”: if you want to see it in “**Design View**” you just need to change its view-type (click *B.4.1.4*).

A **VBA Module** is **always opened** in the specific window of the **VBA editor**.

### B.4.1.4 How do I change the view-type of an opened object?

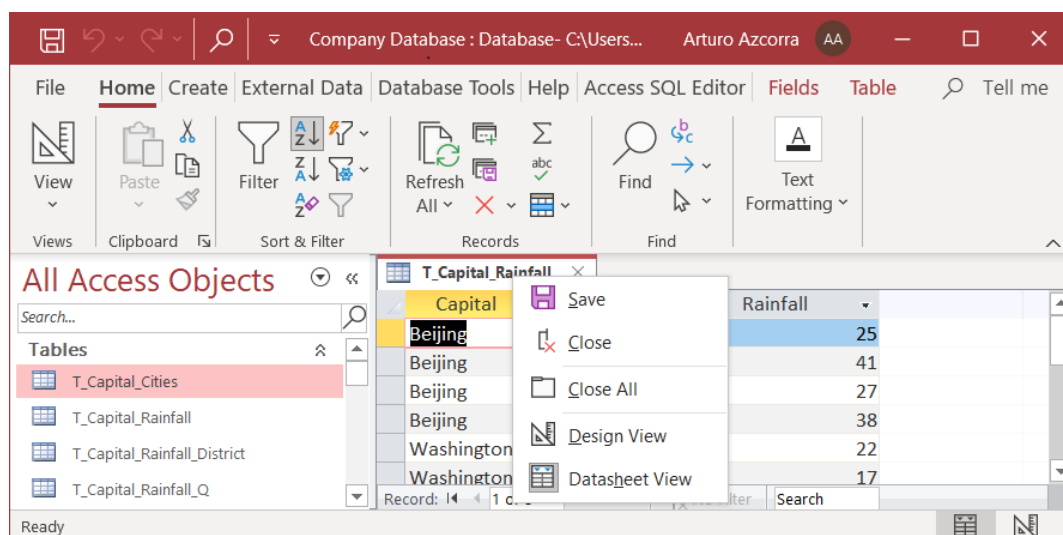
You can **change** the **view-type** of an opened object in either of the following ways:

- Click on either “**Home**” or “**Design**” from the “Ribbon-bar” and look at the **leftmost side** of the Ribbon: if the **view-type icon** that you want (e.g., the **Datasheet View** “” icon), is currently shown, you click on it; otherwise, you click on the “**View**”<sup>View</sup> icon, and then click on the **view-type** that you want from the pop-up menu.
- Click on the **view-type icon** that you want from the “Status-bar”. The “Status-bar” is placed at the **right-bottom** side of the MS-Access window frame. The **view-type icons** that you can click (**depending** on the specific object) are:
  - **Design View** “” icon
  - **Datasheet View** “” icon (except for **Reports**)

<sup>12</sup> If you configure the option “**Single-click**” you will **open** an object with one click (instead of with a double-click), and also, you will **select** an object by just placing the mouse over it (instead of by clicking on it).

<sup>13</sup> For the case of “**Modules**”, there is no “**Open**” option in the pop-up menu, and you will have to click on the “**Design View**” option.

- **SQL View** “SQL” icon (only for **Queries**)
- **Form View** “Form” icon (only for **Forms**)
- **Layout View** “Layout” icon (only for **Forms** and **Reports**)
- **Print Preview** “Print” icon (only for **Reports**)
- Right-click on the object **tab** (at the top of the “Object Area”) and in the pop-up menu click on the **view-type** that you want. The menu you get when you right-click on a **Table tab** is shown in the following screenshot:



In the menu you can see the options “Save”, “Close”, “Close All”, plus, **all** the different views in which **the specific** object (in this case a Table) can be presented.

Notice that **each** object can **only** be viewed in **certain view-types**. The available **view-types** for each specific object are the ones **listed** for **it** in the corresponding bullet point slightly above.

If you **change the view-type** of an object that has **unsaved layout or design changes**, there are two possible cases:

- **MS-Access asks if you want to save the unsaved changes**  
If you click “Yes”, the unsaved changes will be **saved**, and the object will be changed to the new **view-type**. If you click “No”, changes will **not** be saved, and the object will **remain** in the current **view-type**. To change the **view-type** without saving changes, **close** the object (click *B.4.1.7*) and then **open** it (click *B.4.1.3*) in the view type that you want.
- **MS-Access does not ask if you want to save the unsaved changes**  
MS-Access **reminds** that this object has **unsaved changes**. It will therefore **ask** if you want to **save them** at a **later stage**, when you either **close** the object (click *B.4.1.7*) or change it to another **view-type**.

The first case usually happens when you change from “**Design View**” to “**Datasheet View**”, while the second one happens when you change from “**Datasheet View**” to some other **view-type**. However, this is not a complete list of all that can happen.



#### B.4.1.5 How do I view another of the opened objects?

Click on its **object tab** (click *B.2.9*) at the top of the “Object Area” (click *B.2.8*).

If the MS-Access window width is **too narrow** to show **all** the tabs of opened objects, MS-Access will show a **next left** “◀” icon on the **leftmost side** and/or a **next right** “▶” icon on the **rightmost side** of the **object tabs**. The **next left** “◀” icon is shown when there are **hidden tabs** to the **left** of the visible ones. The **next right** “▶” icon is shown when there are **hidden tabs** to the **right** of the visible ones. If you click on the **next left** “◀” icon, the **first hidden** tab on the **left** side will become visible, and the required ones (depending on their width) on the **right** side will become hidden. If you click on the **next right** “▶” icon, the **first hidden** tab on the **right** side will become visible, and the required ones (depending on their width) on the **left** side will become hidden.

By default, “Object tabs” are **ordered** left to right by the moment when the object was opened: the object opened **longest ago is leftmost**, and the object opened **most recently is rightmost**. However, you can change the order of **tabs** by doing drag-and-drop over the tab that you want to move.

#### B.4.1.6 How do I save the layout/design of an opened object?

You **save** the **layout** and/or **design** changes of an **opened** object by right-clicking on its **tab** (at the top of the “Object Area”) and clicking on “**Save**” from the pop-up menu.

If the object was opened in “**Datasheet View**”, you are saving its **layout**.

If the object was opened in “**Design View**”, you are saving its **design**.

For the specific case of **Queries**, if you did **formatting** changes in “**Design View**”, you are saving its **layout**. If you did **design** changes (e.g., change the **field** order) in “**Design View**”, you are saving its **SQL code**. If the Query was opened in “**SQL View**”, you are saving its **SQL code**.

Notice that any data modification (i.e., add records, edit records, or delete records) performed on a Table or Form in “**Datasheet View**” are **directly updated** on the corresponding database Tables, right when you do each of them. Therefore, you **do not have to save** them and actually you **cannot discard** them.

Finally, you can **also save** the **layout** and/or **design** changes of an **opened** object by either **closing** it (click *B.4.1.7*) or **changing** its **view-type** (click *B.4.1.4*), and then clicking “**Yes**” in the dialogue-box where MS-Access asks if you want to **save changes**. However, this is **risky** because you may mistakenly **close** the dialogue box or click “**No**”, therefore **discarding** your unsaved **layout** and/or **design** changes. For this reason, this way of saving is considered a **bad practice**.

#### B.4.1.7 How do I close an opened object?

You **close** an **opened object** in either of the following ways:

- Click on the close icon “**X**” placed at the rightmost side of its **tab** (at the top of the “Object Area”).
- Right-click on its **tab** and click on “**Close**” from the pop-up menu.

You can **close all opened objects** by right-clicking on **any** object tab (at the top of the

“Object Area”) and clicking on “**Close all**” from the pop-up menu.

If you **close** an object that has **unsaved changes**, MS-Access will ask if you want to save changes to the **design** before closing the object. If you click “**Yes**” the unsaved changes will be **saved**; otherwise, they will be lost.

Notice that MS-Access asks to save changes to the object **design**, even if you only did changes to the object **layout** (click *B.4.1.6*). This may be confusing if you are unaware of it, and this is why I am pointing it out.







#### **B.4.1.8 How do I rename an object?**

You **rename** an object by right-clicking (i.e., click with the right mouse button) on its name in the “**Navigation Pane**” and click on “**Rename**” from the pop-up menu. The object name will be selected (shown as shaded) and you can edit its name with the keyboard, pressing the “**Enter**” key when you are done. If you change your mind and do not want to rename the object, press the “**Esc**” key, and the object will **not** be renamed.

**Be aware** that if you set the “**Single-click**” option, (click *B.1.1*) this causes a **somehow annoying side effect** when **renaming** objects. If the “**Single-click**” option is set, **and** the mouse **stays within** the “**Navigation Pane**” **after** you clicked on “**Rename**”, the renaming **will not work**. The reason is that “**Single-click**” implies that **objects** are **selected** by just placing the mouse-pointer over them: therefore, if the mouse-pointer **stays** (after having clicked on “**Rename**”) **within** the “**Navigation Pane**”, the **object name** below the mouse-pointer will be **selected**, and the object name that had just been selected for renaming will be **cleared**. The solution is really simple (just **remind** about it): when you want to rename an object, after having clicked on “**Rename**”, you **must** immediately move the mouse-pointer out of the “**Navigation Pane**”. Once the mouse-pointer is out of the navigation pane, you may edit the object name with the keyboard as described above.

#### **B.4.1.9 How do I run a Query?**

You **run** a Query from MS-Access in either of the following ways:

- (Double<sup>14</sup>) Click on the Query **name** in the “**Navigation Pane**”.
- Right-click on the Query **name** in the “**Navigation Pane**” and click on “**Open**” from the pop-up menu.
- If the Query is open in “**Datasheet View**”, you click on the **Refresh All**  icon from the “**Home**” Ribbon. Do not mistake it with the **Refresh**  icon, that can be shown instead of the **Refresh All**  icon. In case you are seeing the **Refresh**  icon, click on the “**Refresh**”  icon right below it, and click on “**Refresh All**” from the pop-up menu. This will **both run** the Query and change the icon to the **Refresh All**  icon.

---

<sup>14</sup> If you configure the option “**Single-click**” you will **open** an object with one click (instead of with a double-click), and also, you will **select** an object by just placing the mouse over it (instead of by clicking on it).

- If the Query is opened in “**Design View**” or in “**SQL View**”, you change its **view-type** to “**Datasheet View**” (click *B.4.1.4*).

If you are using the plug-in “**Access SQL Editor**” (click *F.5*), you can **also run** a Query directly from it (click *F.5.4.5*).

Regardless of which way you run the Query, its results will be shown<sup>15</sup> in “**Datasheet View**” in its **own** MS-Access “Query pane”, inside the MS-Access object area, and **outside** the “**Access SQL Editor**” pane.

#### **B.4.1.10 How do I select one or more objects?**

You **select one** object from the “**Navigation Pane**” either by clicking on it, or by placing the mouse pointer over its object **name**, depending on setting of the “**Single-click**” option. My advice is you **set** the “**Single-click**” option (click *B.1.1*), in which case you **select one** object by placing the mouse pointer over its **name** in the “**Navigation Pane**”.

You **select** a **range** of contiguous objects by **selecting** (see previous paragraph) **the** object name on **one of the ends** of the object **range**. You then press the “**Shift**” key **while** you **select the** object name of **the other end** of the object range (this is called a “**shift select**” or “**Shift+select**”).

You **select** an **additional** object to the ones already selected by pressing the “**Ctrl**” key **while** you **select the additional** object name (this called a “**control select**” or “**Ctrl+select**”). Notice that you can do this several times, to add several individual objects to the set of already selected objects. Notice also that you can do this after having selected a range of objects.

You **unselect** an object **from** the ones already selected by pressing the “**Ctrl**” key **while** you **select the additional** object name (this called a “**control select**” or “**Ctrl+select**”). Notice that you can do this several times, to **unselect** several individual objects to the set of already selected objects. Notice also that you can do this after having selected a range of objects.

When you have **selected** one or more objects, the **selected object name(s)** will be **highlighted** with **pink background**. You can then **delete** them (click *B.4.1.13*) or **cut/copy** them (click *B.4.1.14*).

#### **B.4.1.11 How do I edit a Query?**

If you are **not** using the plug-in “**Access SQL Editor**” click *F.4.6*.

If you **are** using the plug-in “**Access SQL Editor**” click *F.5*.

#### **B.4.1.12 How do I create an object?**

Depending on the type of “**Navigation Pane**” **object** that you want to **create**, you have to go to different sections, as follows:

- To create a **Table**, click *D.3.1*.
- To create a **linked Table**, click *K.3.6*.
- To create a **Query**, click *F.4.5*.

---

<sup>15</sup> Unless you configure a **non-advisable** option in the “**Access SQL Editor**”.

- To create a **Form**, click *D.10.2*.
- To create a **Report**, click *D.12*.
- To create a **Module**, click *K.9.1*.

Relationships are **not** objects from the “**Navigation Pane**”: if you want to create a Relationship, click *D.9.1*.

In the sections above you will not only find the instructions on how to create each specific object, but also design advice on how to configure it.

#### **B.4.1.13 How do I delete objects?**

To **delete only one** object, you right-click (i.e., click with the right mouse button) on **the** object name in the “**Navigation Pane**” and click on “**Delete**” from the pop-up menu.

To **delete one or more** object(s), you first **select** it/them (click *B.4.1.10*). Then, you either press the “**Supr**” key, or rather, right-click on one **of the selected** object **names** and click on “**Delete**” from the pop-up menu.

Either way MS-Access will ask you if you are sure you want to delete the object(s), and you will have to confirm or cancel.

#### **B.4.1.14 How do I copy/cut and paste objects?**

To **copy/cut only one** object from the “**Navigation Pane**”, you right-click (i.e., click with the right mouse button) on **the** object name in the “**Navigation Pane**” and in the pop-up menu click on “**Copy**” to **copy** it, or on “**Cut**” to **cut** it.

To **copy/cut one or more** object(s), you first **select** it/them (click *B.4.1.10*). Then, you press “**Ctrl-c**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**c**” key) to copy it/them or press “**Ctrl-x**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**x**” key) to **cut** it/them. You can alternatively right-click on one **of the selected** object names and in the pop-up menu click on “**Copy**” to **copy** it/them, or on “**Cut**” to **cut** it/them.

To **paste** the object(s) that you have just cut/copied, you can do it in either of the following ways:

- Right-click on **any object name** in the “**Navigation Pane**” and click on “**Paste**” from the pop-up menu.
- Right-click anywhere on the “**Navigation Pane**” **heading** and click on “**Paste**” from the pop-up menu.
- Place the mouse **anywhere** over the “**Navigation Pane**” and then press “**Ctrl-v**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**v**” key).

**Very important** to note that you can **cut/copy** and **paste** objects **between different database files**.

#### **B.4.2 How do I configure the way object names are grouped, ordered, displayed or hidden in the “Navigation Pane”?**

You may click:

- “*B.4.2.1 How do I configure the way object names are grouped in the “Navigation*”

Pane”?”

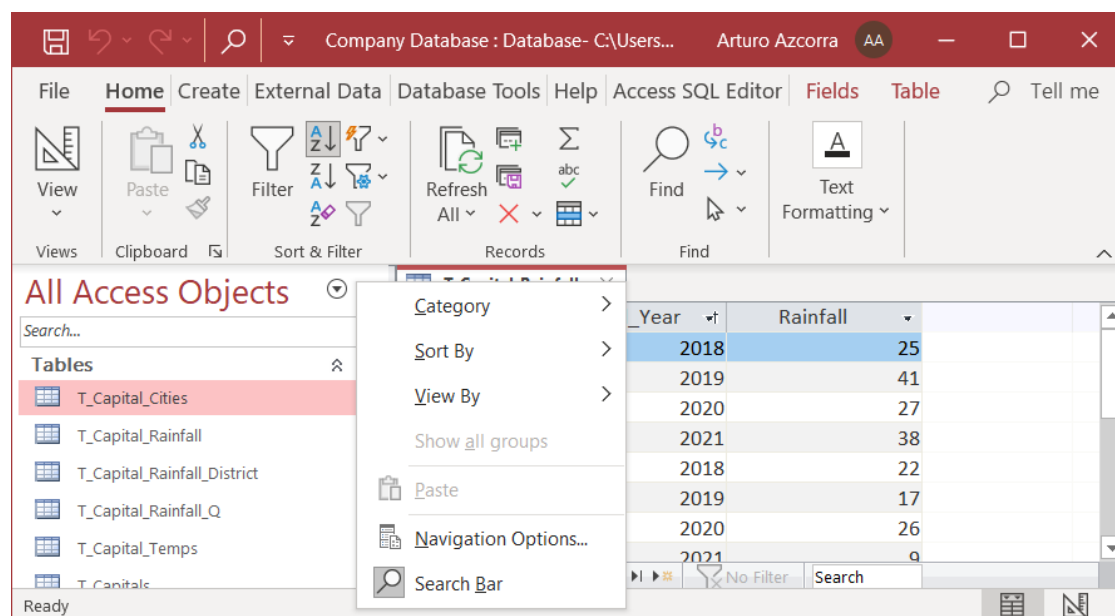
- “B.4.2.2 How do I configure the way object names are ordered in the “Navigation Pane”?”
- “B.4.2.3 How do I configure the way object names are displayed in the “Navigation Pane”?”
- “B.4.2.4 How do I hide/unhide whole object name groups in the “Navigation Pane”?”
- “B.4.2.5 How do I hide/unhide an individual object name in its object group?”
- “B.4.2.6 How do I hide/unhide all system object names?”

### B.4.2.1 How do I configure the way object names are grouped in the “Navigation Pane”?

You can do it in either of the following ways:

- Right-click on the “**Navigation Pane**” heading (click B.4), move the mouse over “**Category**” and on the resulting pop-up menu click on the grouping type you want to select.
- Click on the “**Navigation Pane**” heading (except on the “<<” icon) and click on the grouping type you want to select (grouping types are shown under the label “**Navigate To Category**”).

See in the following screenshot the pop-up menu shown when you right-click on the “**Navigation Pane**” heading:



If you place the mouse pointer over “**Category**” (in the menu shown above), you can select one among the following five **grouping types** that MS-Access offers:

- **Personalized**  
Shows object names grouped as you have manually configured. You can define your own group names and assign each object name to the group name you want in the

dialog box of “**Navigation Options**”. The order of groups is alphabetical by group name. The last group of objects is “**Unassigned Objects**” that includes the names of objects that you have not yet assigned to a specific group. In this case, the default label inside the “**Navigation Pane**” heading is “**Personalized**”.

- **Tables and Related Views**

Shows objects grouped by the Table name they are directly related to. The last group of objects is “**Unrelated Objects**” that includes the names of objects not directly related to any Table. In this case, the default label inside the “**Navigation Pane**” heading is “**All Tables**”.

- **Object Type (default grouping)**

Shows objects grouped by object type, in the following top-down order of groups: Tables, Queries, Forms, Reports and Modules. In this case, the default label inside the “**Navigation Pane**” heading is “**All Access Objects**”.

- **Created Date**

Shows objects grouped by their object creation date, in the following top-down order of groups: Today, Yesterday, Two Days Ago, Last Week, Two Weeks ago, Last Month and Older. In this case, the default label inside the “**Navigation Pane**” heading is “**All Dates**”.

- **Modified Date**

Shows objects grouped by their date of last modification, in the following top-down order of groups: Today, Yesterday, Two Days Ago, Last Week, Two Weeks ago, Last Month and Older. In this case, the default label inside the “**Navigation Pane**” heading is “**All Dates**”.

#### **B.4.2.2 How do I configure the way object names are ordered in the “**Navigation Pane**”?**

Right-click anywhere on the “**Navigation Pane**” heading, move the mouse over “**Sort By**” and click on the order type you want to select.

You can select **two** different **aspects** of the way object names are ordered within each object group. The **first aspect** you can select is whether object names are ordered within each object group in **ascending** or **descending** order. The **second aspect** you can select is the **criterion** to **order** the object names.

You can select one among the following **five ordering criteria** that MS-Access offers:

- **Name (default)**

Objects are ordered within each object group alphabetically on the **object name**.

- **Type**

Objects are ordered within each object group based on the **object type** (taking also into account the Query type).

- **Created Date**

Objects are ordered within each object group based on the **creation date** of each object.

- **Modified Date**

Objects are ordered within each object group based on the **date of last modification**

of each object.

- **Remove Automatic Sorts**

Objects are ordered within each object group as the user has **manually configured**. If you have selected this option, you can change the order of an object within each object group by doing drag-and-drop over the object.

### **B.4.2.3 How do I configure the way object names are displayed in the “Navigation Pane”?**

Right-click on the “**Navigation Pane**” heading, move the mouse over “**View By**” and click on the viewing type you want to select.

You can select one among the following three viewing types that MS-Access offers:

- **List** (default)

Each **object name** is displayed with an object **type small icon** on its **left** side.



- **Icons**

Each **object name** is displayed with an object **type large icon** on its **left** side. I do not recommend this view because is the same as “list”, but you can see fewer objects on the screen due to the larger size of icons.

- **Details**

Each **object name** is shown with an object **type large icon** on its **left** side, the object **type** in text (“**Table**”, “**Query**”, “**Form**”, “**Report**” or “**Module**”) on its **right** side, and the object **creation date** plus the object **last modification date** below the object name.

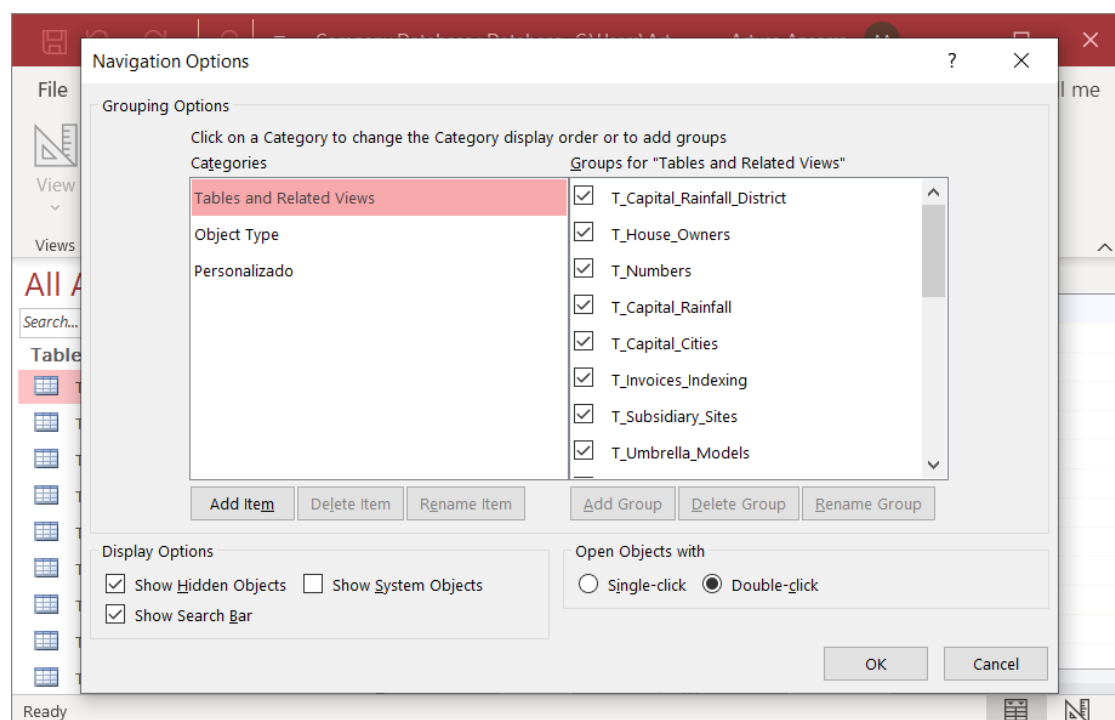
### **B.4.2.4 How do I hide/unhide whole object name groups in the “Navigation Pane”?**

To hide/unhide **a given object name group**, click on the **hide** “” icon or on the **unhide** “” icon, that is placed on the right side of each object group heading. Notice that in this case the object group heading itself is **not hidden**, but all its object names are hidden.

**Another way** to hide/unhide **a given object name group**, is right-clicking on the “**Navigation Pane**” heading and click on “**Navigation Options**” from the pop-up menu. This will show a sub-window where you can see three object grouping types (“**Tables and Related Views**”, “**Object Type**” and “**Personalized**”). Select the grouping type that you want, and then **tick/untick each individual object group** that you want to unhide/hide. Notice that in this case the object group heading is **hidden** in addition to all its object names.



The next screenshot shows the sub-window that will appear when you click on “Navigation Options”:



To **hide all the object name groups except one**, click anywhere on the “Navigation Pane” heading (except on the “<<<” icon) and in the pop-up menu (under the heading “Filter By Group”) click on **the group you want to see**. When you do this, the “Navigation Pane” heading will **change to the group heading** of the only group that you have selected to be shown.

You can **unhide all the object name groups** in either of the following ways:

- Click on the “Navigation Pane” heading (except on the “<<<” icon) and click on “All the Access Objects” placed at the bottom of the pop-up menu.
- Right-click on the “Navigation Pane” heading and click on “Show all groups” from the pop-up menu.

When you unhide all object name groups, the “Navigation Pane” heading will change to the default heading of the **object name grouping type** that you are using at this moment.

#### B.4.2.5 How do I **hide/unhide** an individual **object name** in its **object group**?

MS-Access allows you to hide an object name by right-clicking on its name and clicking on “Hide in this Group” from the pop-up menu. Hidden objects are **not shown** in its corresponding object group, unless you set the option “Show Hidden Objects”.

In case you need to **see all hidden objects** (e.g., to browse through them and/or to one/modify one or more) you can make MS-Access show all of them by setting the option “Show Hidden Objects”. You can **unset** this option when you are done handling the objects you wanted and will all stop being shown. You can set or unset this option

in either of the following ways:

- Right-click on the “**Navigation Pane**” heading and click on “**Navigation Options...**” from the pop-up menu.
- Click the sequence “**File**→**Options**→**Current Database**”. This will show a sub-window where you should scroll until you see the heading “**Navigation Options**”, and there click on the button “**Navigation Options...**”.

Regardless of which way you use, a sub-window will open, and on its bottom-left side you will see the option “**Show Hidden Objects**” with a checkbox on its left side. By **ticking** or **unticking** that checkbox MS-Access **unhides** or **hides all** objects marked as hidden. After having ticked/unticked the checkbox, you should click on the “**OK**” button to hide this “**Navigation Options**” sub-window.

Notice that when hidden objects are shown their name is **shadowed**, to distinguish them from unhidden objects.

If you want to **permanently unhide** a specific object, set the option “**Show Hidden Objects**” (as indicated above), right-click on the object name and click on “**Unhide in this Group**” from the pop-up menu. Then, you have to unset the option “**Show Hidden Objects**” to hide again all hidden objects.

#### **B.4.2.6 How do I hide/unhide all system object names?**

You can hide/unhide all system object names by setting/unsetting the option “**Show System Objects**”. You can set/unset this option in either of the following ways:

- Right-click on the “**Navigation Pane**” heading and click on “**Navigation Options...**” from the pop-up menu.
- Click the sequence “**File**→**Options**→**Current Database**”. This will show a sub-window where you should scroll until you see the heading “**Navigation Options**”, and there click on the button “**Navigation Options...**”.

Regardless of which way you use, a sub-window will open, and on its bottom-left side you will see the option “**Show System Objects**” with a checkbox on its left side. By **ticking** or **unticking** that checkbox “MS-Access **unhides** or **hides all system objects**”. After having ticked/unticked the checkbox, you should click on the “**OK**” button to hide this “**Navigation Options**” sub-window.

The unhidden system objects will be shown in their corresponding object group. System object names will be **shadowed** to indicate that their built-in configuration is as hidden.

### **B.5 What is a Table/Query/Form in “Datasheet View”?**

A Table or Form in “**Datasheet View**” allows to **view**, **enter**, **delete** and **modify** their **records**. A Query in “**Datasheet View**” allows to **view** the Query **results**.

If you want to **create** and **configure** Tables, click *D.3*.

If you want to **create** and **configure** Queries, click *F.4.5*.

If you want to **create** and **configure** Forms, click *D.10*.

Coming back to a “Table/Query/Form pane” in “**Datasheet View**”, it shows each **record**

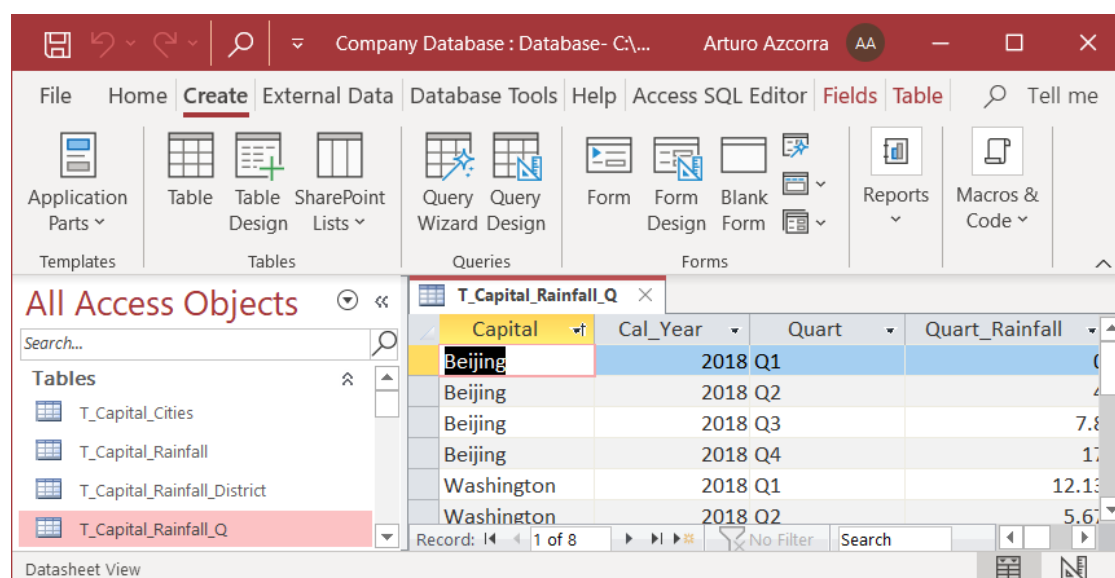
as a **row**, shows the **field names** as **columns** (with the field name as the column heading), and shows the **field values** of each record as a **value** in a **cell** (row and column).

If **all** the rows and/or columns **do not fit** in the vertical and/or horizontal size of the “Table/Query/Form pane”, then a vertical and/or a horizontal **scrollbar** will be shown. The vertical scrollbar will be placed at the leftmost side of the pane. The horizontal scrollbar will be placed at the bottom of the pane, right above the “Navigation bar”.

At the very bottom of the “Table/Query/Form pane” you see the “**Navigation bar**”:



You may see the elements I have just described in the following screenshot of a “Table pane” in “**Datasheet View**”:



In a “Table pane” (not in “Query pane” nor in a “Form pane”) it is possible that you also see an additional column with the heading “**Click to Add**”. If this happens, I advise you remove this column because I consider it a bad practice (click *B.1.1*).

You may **sometimes** see a “+” sign to the left of the records, where you can click to show sub-Tables of related values. My advice is you do not use this functionality because it sometimes fails and creates problems.

You may click:

- “*B.5.1 What is the “Navigation bar” in “Datasheet View”?*”
- “*B.5.2 How do I select a range of fields/records in “Datasheet View”?*”
- “*B.5.3 How do I select one field’s value in “Datasheet View”?*”
- “*B.5.4 What are the differences between selecting one field and one field’s value in “Datasheet View”?*”
- “*B.5.6 How do I manage a Table/Query/Form “Property Sheet” in “Datasheet View”?*”

### B.5.1 What is the “Navigation bar” in “Datasheet View”?

The “Navigation bar” serves to **go to the record** you want, to **know in which record you are**, and to **create a new record**. The “Navigation bar” is extremely useful to find your way around in a Table/Query/Form with lots of records. The next screenshot shows the “Navigation bar”:



The “Navigation bar” has the following elements:

- Position box** “1 of 28”
 

Shows the **row number** of the currently **selected record** and the **total number** of records in the Table/Query/Form. In the screenshot you can see “1 of 28” indicating that the currently selected record is number 1 out of a total of 28 records.
- Goto buttons** (“|◀” and “▶|”)
 

To go to the **first** record in the Table click on “|◀”, to go to the **previous** record to current one click on “◀”, to go to the **next** record to current one click on “▶”, and finally, to go to the **last** record click on “▶|”. These icons are placed to the left and right of the position box. They will be **shadowed** when **cannot** be applied.
- Create new record button** “▶\*”
 

Clicking “▶\*” moves you to the last row of the “Table/Form/Query pane”, which is the row that serves to create new records.

In a “Query pane” that cannot add records (click *K.4.4*), the “create new record” button “▶\*” will be **shaded** and will **not work**, which is what you typically want.

If the source database file is **read-only**, the “create new record” button “▶\*” will be **shaded** and will **not work** (click *L.8.2*).
- Toggle filter** “No Filter” or “Filtered”
 

Shows if the records displayed in the Table/Query/Form are filtered or not. Also, clicking on it allows you to toggle between showing all the records (i.e., no filter), or showing only the filtered records. The filters are configured using the filter icons from the “Sort & Filter” Ribbon group from the “Home” Ribbon.
- Search box** “Search”
 

Finds records using **incremental substring** match. You just **type-in the text** you are looking for and MS-Access will find (while you type) the first field (cell) that includes that string. If you are done typing-in the searched text and you want to move to the **next match**, press the “Enter” key. The search box is placed on the rightmost side of the “Navigation bar”.

### B.5.2 How do I select a range of fields/records in “Datasheet View”?

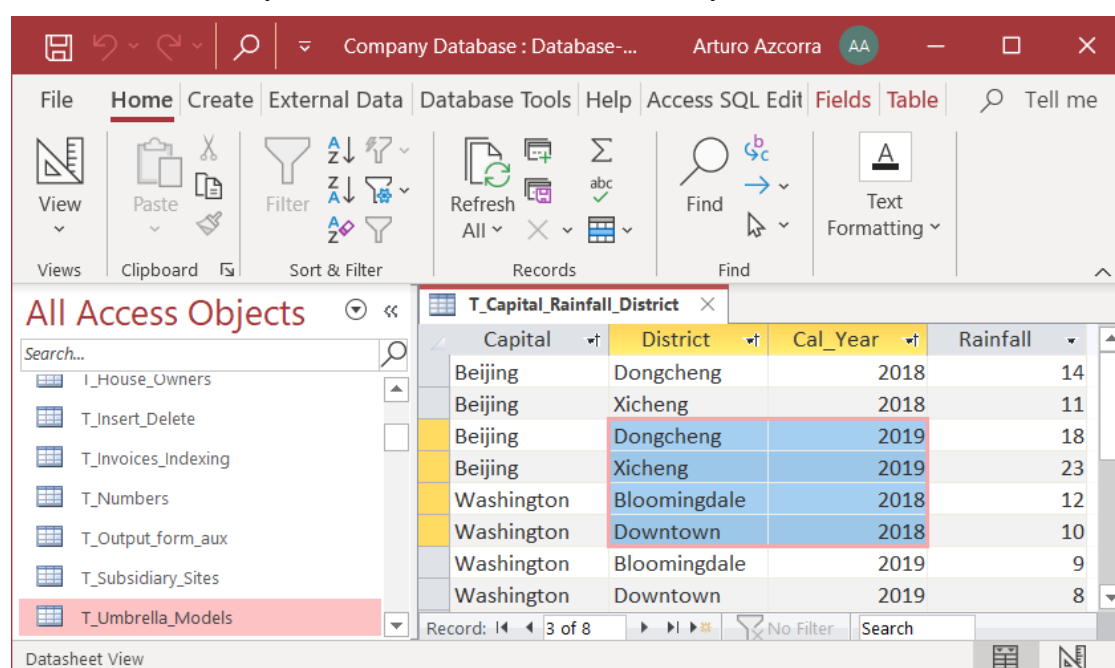
In order to **do different operations** (e.g., copy, paste, delete, ...) over **the fields** (cells) of a Table/Query/Form in “**Datasheet View**”, you will need **first** to **select** the **range of fields** (cells) that you want to act upon. After you have selected the **range of fields** that you want, you can actually do the **operation** (e.g., copy, paste, delete, ...) over the fields.

MS-Access provides the following **very flexible** ways to **select the range of fields** (cells) that **you want** from a Table/Query/Form in “**Datasheet View**”:

- Select **all the rows and columns** (i.e., all the Table’s records with all their fields).

- Select **one row** of fields (i.e., one record, with all its fields).
- Select **one column** of fields (i.e., one field value-list, i.e., a field name across all the records).
- Select **one field** (i.e., one field in one record. Notice this is **different** from selecting **one field's value**, click *B.5.4*).
- Select a range of **contiguous rows of fields** (i.e., a range of records, each with all its fields).
- Select a range of **contiguous columns of fields** (i.e., a range of field names for all the records).
- Select an arbitrary **rectangular range of fields** (i.e., some contiguous fields from some contiguous records).

The following screenshot shows an arbitrary **rectangular range of selected fields** (cells). See how they are shaded in blue to indicate they have been selected:



To use any of the **selection options** in the list above, you first **open** the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

You then use the **selection option** that you want, among the following ones.

### How do I select all the rows and columns?

This is, selecting **all** the Table/Query/Form records with **all** their fields.

Click on the top-left corner-box of the pane (it is the corner-box right below the object icon in the **tab**, that has a small shaded triangle). This will cause MS-Access to **select** the **complete** record-list (**including everything outside** of the current viewing area).

The **Table/Query/Form** should now have a **pink** border (all around, not every cell) and **all** its cells should have **light blue** background, thus showing that they are actually selected.

### How do I select one row with all its fields?

This is, **one** record with **all** its fields.

Click on the box shaded in gray, placed to the left of the record you want to select (i.e., the box at the left of the first field).

The **record** that you wanted to select should now have a **pink** border and **all** its cells should have **light blue** background, thus showing that they are actually selected.

### How do I select one column of fields?

This is, **one** field value-list. In other words, all the fields corresponding to **one** field name in **all** the records in the Table.

Click on the field name, placed in the cell at the top of the column you want to select.

The **column** that you wanted to select should now have a **pink** border and **all** its cells should have **light blue** background, thus showing that they are actually selected.

### How do I select one field?

This is, **one** field of **one** record. Notice that selecting **one field** is **different** from selecting **one field's value** (click [B.5.4](#)).

Place the mouse within the **field** you want to select. Now you will have to move the mouse slowly to place it near the borders of the field, but still being inside. A good approach is to move it near the leftmost side of the field. You have to slowly move it until the mouse pointer changes to a **white cross** “+”. It may take you a little the first time you try, but believe me, the white cross **will appear**. The next screenshot shows what you should see:

City	District	Cal_Year	Quart	T
Beijing	Dongcheng	2018	Q1	
Beijing	Dongcheng	2018	Q2	
Beijing	Dongcheng	2018	Q3	
Beijing	Dongcheng	2018	Q4	
Beijing	Xicheng	2018	Q1	
Beijing	Xicheng	2018	Q2	

While the mouse pointer is still a white cross “+”, do a mouse click.

The **field** that you wanted to select should now have a **pink** border and a **light blue** background, thus showing that they are actually selected.

### How do I select a range of contiguous rows of fields?

This is, a range of records, each with all its fields.

You can select a row range with either of the two following options:

- **Click-and-drag rows**

Place the mouse within the box shaded in gray placed at the left of one of the rows **at either end of row range** you want to select. Press the **right** mouse-button, and also, **without releasing the button, drag** the mouse and place it **over the row at the other end** of the range you want to select. You now release the mouse button, and the row range is selected. Notice that you can select rows **outside of the initial viewing area**, because MS-Access will automatically scroll the rows when you drag the mouse outside of the viewing area.



- **Select and Shift+select rows**

Select one of the rows placed **at either end of the range** you want to select (see above how to **select one row**). Now select the row placed at the **other end of the range, but**, when you do the mouse-click to select it, **do the click while pressing the Shift key** (a so-called “**Shift+click**”). In case you want to select rows **outside of the initial viewing area**, scroll down/up until you can see the row placed at the other end of the range you want.

The **row range** that you wanted to select should now have a **pink** border and **all** its cells should have **light blue** background, thus showing that they are actually selected.

### **How do I select a range of contiguous columns of fields?**

This is, the fields of a **range** of contiguous **field names** for **all** the records.

You can select a column range with either of the two following options:

- **Click-and-drag columns**

Place the mouse within the field with the field name placed at the top of the column placed **at either end of the column range** you want to select. Press the **right** mouse-button, and also, **without releasing the button, drag** the mouse and place it **over the column** placed **at the other end** of the range you want to select. You now **release** the mouse button, and the column range is selected. Notice that you can select columns **outside of the initial viewing area**, because MS-Access will automatically scroll the columns when you drag the mouse outside of the viewing area.

- **Select and Shift+select columns**

Select one of the columns placed **at either end of the range** you want to select (see above how to **select one column**). Now select the column placed at the **other end of the range, but**, when you do the mouse-click to select it, **do the mouse-click while pressing the Shift key** (a so-called “**Shift+click**”). In case you want to select columns **outside of the initial viewing area**, scroll right/left until you can see the column placed at the other end of the range you want.


The **column range** that you wanted to select should now have a **pink** border and **all** its cells should have **light blue** background, thus showing that they are actually selected.

### **How do I select an arbitrary rectangular range of fields?**

This is, some contiguous fields from some contiguous records.

You can select a **rectangle of fields** with either of the two following options:

- **Click-and-drag fields**

Place the mouse within the **field** placed in **one of the four corners of the rectangle** you want to select. Now you will have to move the mouse slowly to place it near the borders of the field, but still being inside. The best approach is to **move the mouse near the leftmost side of the field**. You have slowly move it until the mouse pointer changes to a **white cross** “Page 56 of 725



can select fields **outside of the initial viewing area**, because MS-Access will automatically scroll the columns and rows when you drag the mouse outside of the viewing area.

- **Select and Shift+select fields**

Select the **field** placed in **one of the four corners of the rectangle** you want to select (see above how to **select one field**). Now place the mouse within the field placed at the **opposite corner of the rectangle of fields**, and also, **do a mouse-click while pressing the Shift key** (a so-called “**Shift+click**”). In case you want to select fields **outside of the initial viewing area**, scroll right/left and/or up/down until you see the field placed at the opposite corner of the rectangle of fields you want.

The **rectangle of cells** that you wanted to select should now have a **pink** border and **all** its cells should have **light blue** background, thus showing that they are actually selected.

As you have noticed along the explanation in this section, the way to do the different selection types follows some common principles and should be reasonably easy to remember.

### B.5.3 How do I select one field’s value in “Datasheet View”?

You select one field’s value in either of the following ways:

- **Clicking** while the mouse pointer is **within** the field, but **not** near its borders  
This will select one field’s value.

This will also place the “**type-in cursor**” (a vertical bar “|” blinking for a while) where you clicked, within the **string** representing the field value.

If you rather place the mouse pointer **within** the field and **near its borders**, the mouse pointer becomes a white cross “**+**”, and when clicking you would select one field instead of one field’s value. If you want to know the difference, you may click “*B.5.4 What are the differences between selecting one field and one field’s value in “Datasheet View”?*”).

- **Pressing** the “**Enter**” key or the “**Tab**” key  
If you press the “**Enter**” key or the “**Tab**” key, you will select the field’s value of the field placed to the right of the currently selected field.

This will also **select** the whole **string** representing the field value: this is highlighted by showing the **string** representing the field value in **white** font over **black** background.

- **Pressing** the up-arrow “**↑**” key, the down-arrow “**↓**” key, or the **page** keys  
If you press the up-arrow “**↑**” key or the down-arrow “**↓**” key, you will select the field’s value of the field placed up or down (respectively) to the currently selected field. Likewise, if you press the “**Page up**” key or the “**Page down**” key, you will select the field’s value of the field placed one page up or one page down (respectively) to the currently selected field.

This will also **select** the whole **string** representing the field value: this is highlighted by showing the string representing the value in **white** font over **black** background.

- **Pressing** the **right/left arrow** keys  
Pressing the left-arrow “**←**” key or right-arrow “**→**” key is slightly **tricky**, because

it has different effects, depending on the **location** of the **type-in cursor** “|” and on whether you did any editing on the field or not:

- If the whole **string** representing the field value is selected (i.e., it is shown with **white** font over **black** background), and you did not edit the field:  
When you press the left-arrow “←” key or the right-arrow “→” key you will select the **field’s value** of the field placed to the left or right (respectively) of the currently selected field.
- If the type-in cursor “|” is **within** the string representing the field value:  
When you press the left-arrow “←” key or the right-arrow “→” key you will **move** the type-in cursor **one character** left or right (respectively) **within** the **string** representing the field value. This is useful when you are editing the **string** representing the field value.
- If the type-in cursor “|” is **at the end** (rightmost) of the **string** representing the field value:  
When you press the left-arrow “←” key you will move the type-in cursor **one character** to the left. However, if you press the right-arrow “→” key you will select the **field’s value** of the field placed to the right of the currently selected field.
- If the type-in cursor “|” is **at the beginning** (leftmost) of the **string** representing the field value:  
When you press the right-arrow “→” key you will move the type-in cursor **one character** to the right. However, if you press the left-arrow “←” key you will select the **field value** of the field placed to the left of the currently selected field.

Note the following relevant remarks:

- If the currently selected field is the **rightmost** one of the record and according to the bullets above you select the **field’s value** to its **right**, you will actually select the **field’s value** of the **leftmost** field of the record placed **below**.
- If the currently selected field is the **leftmost** one of the record and according to the bullets above you select the **field’s value** to its **left**, you will actually select the **field’s value** of the **rightmost** field of the record placed **above**.
- If part of the **string** representing the value is **selected** for editing (**white** font over **black** background), the type-in cursor “|” will be placed as follows. If you selected doing double-click, it will be placed in the rightmost side of the selection. If you selected doing click-and-drag, it will be placed where you released the mouse button.

If you correctly selected the **field’s value**, the field’s **border** should now be highlighted in pink color, and the background of the field should be **white** or **light gray** (and should **not** be light blue).

Notice that when you select a **field’s value** the value **shown** may change: if you want to know more about this, you may click “[E.2.3 Why is a value shown in a different way when the Table/Query/Form field’s value has been selected?](#)”.

Notice that selecting **one field** is different from selecting **one field’s value**. If you want to know more about the differences, you may click “[B.5.4 What are the differences between selecting one field and one field’s value in “Datasheet View”?](#)”.

### **B.5.4 What are the differences between selecting one field and one field's value in "Datasheet View"?**

Selecting **one field** (click *B.5.2*) selects the **complete value** of the field, and also, the **field's metadata** (e.g., data type and format).

Selecting **one field's value** (click *B.5.3*) selects the **value** of the field, represented as a **string** that you can edit.

Selecting **one field** is indicated by highlighting the field border with **pink** color, and the field background with **light blue** color. The value shown in the field does not change.

Selecting **one field's value** is indicated by highlighting the field border with **pink** color, but the field background stays with the same color it had (**white** or **light gray**). The value **shown** in the field will change if the field formatting is different from the formatting used in the **string** representing the value. What you see inside the field is the **string** representing its value. Depending on how you selected the **field's value** you may see the **string** representing its value in **black** font over **white/gray** background, or in **white** font over **black** background. The part of the string that is shown in **white** font over **black** background is selected for editing.

Selecting **one field** is done by clicking within the field, when the mouse is near the field border, and the mouse icon becomes a white cross "☒").

Selecting **one field's value** is done by clicking within the field, when the mouse is **not** near the field border, and the mouse is **not** a white cross "☒". Notice there are other ways of selecting **one field's value** (click *B.5.3*).

When you select **one field**, you are selecting its **metadata** (field type and formatting) **in addition** to its **value**. Therefore, if you copy and paste it, you will paste the field **value**, its **data type** and its **formatting** (provided that the target application is compatible, like for example Excel).

When you select **one field's value**, you are **only** selecting the field's **value** itself. Therefore, if you copy and paste it, you will only **paste** the field value **as text**.

When you select **one field**, you are selecting the whole field itself. Therefore, if you press a key (e.g., "9" or "Supr"), the value of the field will be **overwritten**<sup>16</sup> by the key you have pressed (i.e., the field value will now be "9" or blank, respectively), and the field will change to being selected as a **field's value**.

When you select **one field's value**, you are only selecting the field **value** itself, represented by a **string**. Therefore, if you press a key (e.g., "9"), the result will depend on whether part of the **string** representing the field value is selected (i.e., with **white** font over **black** background) and on the position of the type-in cursor "|". To know more about **editing** the **string** representing the field value, you may click *E.2.2*.

You can select **one field** and also a **range** of **fields** (click *B.5.2*).

Conversely, you can **only** select **one field's value**, and you **cannot** select a **range** of **field's values**.

---

<sup>16</sup> If the field has a drop-down menu configured, it may behave differently: pressing a character key when the **field** is selected will select the **field's value**, instead of overwriting the value of the field.

### B.5.5 How do I find a record in a Table/Query/Form “Datasheet View”

You do it in either of the following ways:

- Type-in the string you are looking for in the **Search box** “

You can type the searched string in the “**Find What:**” box, and configure the corresponding options using the drop-down menus and checkboxes.

- Use the **vertical scrollbar** on the right side of the “Table/Query/Form pane” to manually find the record, combining this with sorting the records (click *H.2.3*). This is somehow cumbersome but may work well for a small number of records.

### B.5.6 How do I manage a Table/Query/Form “Property Sheet” in “Datasheet View”?

A **Table** or **Query** in “**Datasheet View**” does not have a “**Property Sheet**”.

A **Form** in “**Datasheet View**” has a “**Property Sheet**”, but it is **the same** as the **Form** “**Property Sheet**” in “**Design View**”. If you want to know how to configure the “**Property Sheet**” of a **Form**, you may click “*B.8.2 How do I configure a Form’s “Property Sheet” in “Datasheet View” or “Design View”?*”.

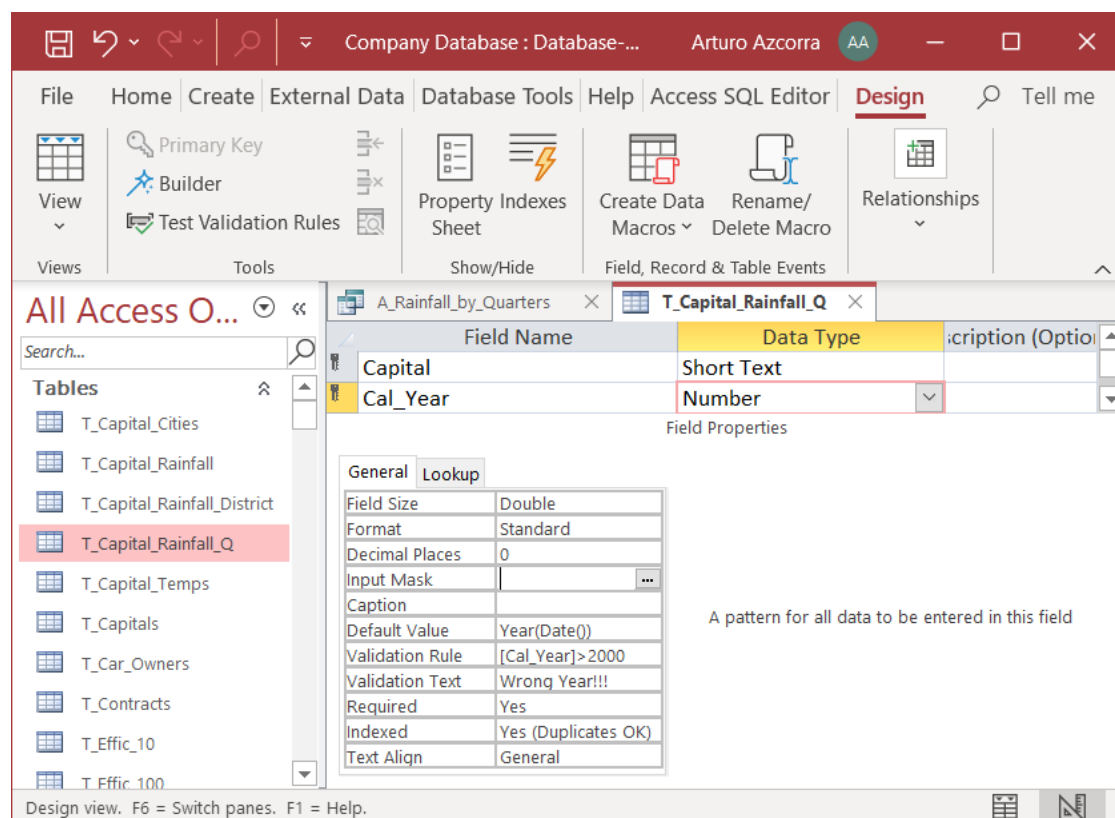
## B.6 What is a Table in “Design View”?

A **Table** in “**Design View**” allows to **configure** its **fields**, the **field’s properties** and the **Table’s properties**.

If you want to **create** and **configure** **Tables**, click *D.3*.

Coming back to a “Table pane” in “**Design View**”, you will see two different parts: the **top sub-pane** and the **bottom sub-pane**. On the **top sub-pane**, you see the **list of fields**

of the Table. On the **bottom sub-pane**, you see the **properties** of the **selected field**. See the next screenshot:



The **top sub-pane** shows the Table **fields** (one in each row). For each Table field name (i.e., each **row**), it contains (left to right) the **name of the field**, the **data type of the field**, and the optional **description** (free text) of the field. In the screenshot above you may see two rows: the first (top) row represents a field named “Capital” with **Short Text** data type and no description; the second row represents a field named “Cal\_Year” with **Number** data type and no description.

The **bottom sub-pane** shows the **field properties** of the **field** that is **currently selected** (the field that is highlighted in the top sub-pane). In the screenshot above the currently selected field is “Cal\_Year”.

The bottom sub-pane has **two tabs**: “**General**” and “**Lookup**”. The one shown by default is the “**General**” tab. You can show the tab you want by clicking on the corresponding tab name “**General**” and “**Lookup**”.

If you want to know how to view the Table fields and their “**General**” and “**Lookup**” **properties**, you may click:

- “B.6.1.1 How do I view the Table fields in “Design View”?”
- “B.6.1.2 What are a Field’s “General” tab properties in “Design View”?”
- “B.6.1.3 What are a Field’s “Lookup” tab properties in “Design View”?”

If you want to configure the **indexes** of a Table, you may click:

- “B.6.2 How do I manage Table indexes in “Design View”?”

If you want to configure a Table's "**Property Sheet**", you may click:

- "B.6.3 How do I unhide/hide a Table "Property Sheet" in "Design View"?"
- "B.6.4 How do I configure a Table "Property Sheet" in "Design View"?"

If you want to **create** and **configure** Tables, you may click:

- "D.3 How do I create and design a Table and its fields?"
- "D.4 How do I configure a Table field data type and size?"
- "D.5 How do I configure a Table field validation rule, indexing, and other properties?"
- "D.6 How do I configure the Primary Key field(s) of a Table?"
- "D.7 How do I add simple and/or composite index(es) to a Table?"
- "D.8 How do I configure the properties of a Table?"

### **B.6.1 How do I view, add, delete and configure my Table fields?**

You may click:

- "B.6.1.1 How do I view the Table fields in "Design View"?"
- "B.6.1.2 What are a Field's "General" tab properties in "Design View"?"
- "B.6.1.3 What are a Field's "Lookup" tab properties in "Design View"?"
- "B.6.1.4 How do I select one or more Table fields in "Design View"?"
- "B.6.1.5 How do I add Table fields in "Design View"?"
- "B.6.1.6 How do I copy/cut and paste Table fields in "Design View"?"
- "B.6.1.7 How do I reconfigure my Table fields in "Design View"?"
- "B.6.1.8 How do I delete Table fields in "Design View"?"
- "B.6.1.9 How do I reorder Table fields in "Design View"?"

#### **B.6.1.1 How do I view the Table fields in "Design View"?**

If the Table is not opened in "**Design View**", you either **open** it (click B.4.1.3) in "**Design View**" or **change** its **view-type** (click B.4.1.4) to "**Design View**".

You now **view all** the Table fields on the **top sub-pane**. Each field is in **one** row. Within each **row**, you see (left to right) the field **name**, the field **type** **and** the optional field **description** (click B.6).

Clicking anywhere within a **field row** in the **top sub-pane** will **show its properties** in the **bottom sub-pane**. There are **two** property **tabs**: the "**General**" properties **tab** (click B.6.1.2) and the "**Lookup**" properties **tab** (click B.6.1.3).

You may also view the Table's "**Property Sheet**" (click B.6.3).

Once you are done viewing your fields, you may **close** the Table (click B.4.1.7) or change it to "**Datasheet View**" (click B.4.1.4).

### B.6.1.2 What are a Field's "General" tab properties in "Design View"?

The "**General**" tab contains field properties **other** than the ones to configure how to enter data (see next subsection). Some examples of these properties are "**Field Size**", "**Required**" or "**Indexed**".

The "**General**" tab properties of a field **depend on the field's data type**. You will therefore see **different** properties if you select another field with a different data type, or, if you change the data type of the current field. If you want to know more about this, you may click:

- *"D.5 How do I configure a Table field validation rule, indexing, and other properties?"*.

### B.6.1.3 What are a Field's "Lookup" tab properties in "Design View"?

The "**Lookup**" tab contains field properties to configure how to enter data in the field when the Table is in "**Datasheet View**".

The "**Lookup**" tab properties of a field **depend on the field's data type**. You will therefore see **different** properties if you select another field with a different data type, or, if you change the data type of the current field. If you want to know more about this, you may click:

- *"D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?"*
- *"K.1.8 What are good practices in configuring my drop-down menus?"*

### B.6.1.4 How do I select one or more Table fields in "Design View"?

If the Table is not opened in "**Design View**", you either **open** it (click *B.4.1.3*) in "**Design View**" or **change** its **view-type** (click *B.4.1.4*) to "**Design View**".

You can then **select** field rows as follows:

- To **select one field row**  
Click on the box shaded in gray, placed to the left of the field name you want to select. This is **the same** process as for selecting rows in a Table or Query result. The **border** of the **complete** field row (not only one cell) should be **pink** and the **square** to the left of the field name should be **dark gray**: these two highlights show that this field row has been selected.
- To **select a range of contiguous field rows**  
This is **the same** process as for selecting a range of contiguous rows in a Table or Query result. You can select a field row range in either of the following ways:
  - **Click-and-drag field rows**  
Place the mouse within the box shaded in gray placed at the left of one of the field rows **at either end of row range** you want to select. Press the **right** mouse-button, and also, **without releasing the button, drag** the mouse and place it **over the field row at the other end** of the range you want to select. You now release the mouse button, and the field row range is selected.
  - **Select and Shift+select field rows**  
Select one of the field rows placed **at either end of the range** you want to select



(see above how to **select one field row**). Now select the field row placed at the **other end of the range, but**, when you do the mouse-click to select it, **do the click while pressing the Shift key** (a so-called “**Shift+click**”).


Regardless of how you did it, the **border** of the **whole range** of field rows should be **pink** and the **square** to the left of **all** the field names of the **range** should be **dark gray**: these two highlights show that this range of field rows has been selected.

- To **select an arbitrary set of field rows**  
**Select and Ctrl+select arbitrary field rows**

Select one of the field rows of the set you want to select (see above how to **select one field row**). Now select **another** field row of the set, **but**, when you do the mouse-click to select it, **do the click while holding down the “Ctrl”**<sup>17</sup> key (a so-called “**Ctrl+click**”). Do this as many times as you want.

Once you have finished doing this, **all** the field rows in the **selected set** should have the box to the left in **dark gray** and the **last** field row you selected will its border highlighted in pink color. The **dark gray** boxes tell you that **all** that field rows have been selected.

When you select an **arbitrary** set of rows (this is, not a range of contiguous field rows) in a Table in “**Design View**” you can **only** use this arbitrary set to configure the **Key** fields. If you try to do **any other** action (e.g., move or delete a selected) different from defining the **Key** fields, it will not work.

**Selecting Table fields** in “**Design View**” is very useful to **do different operations** (e.g., move, delete, set as **Key** fields, ...) over **them**. After you have selected the **field rows** that you want, you click the command to do the actual **operation** (e.g., move, delete, set as **Key** fields, ...) over the selected field rows. For example, if you wanted to configure the **Primary Key** fields, you could now click on “**Design**” from the “**Ribbon-bar**” (below the “**Table Tools**” contextual label) and then on the **Primary Key**  icon, and all the fields that you selected will be configured as **Key** fields.

#### **B.6.1.5 How do I add Table fields in “Design View”?**

If the Table is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

You can then **add** field row(s) to the Table in either of the following ways:

- **Click on** the “**Field name**” cell of **any** blank row (below the last row of the current fields) and type-in the field’s name. Click on the “**Field type**” cell of the same field row, and type-in or select with the menu the field type. Click on the “**Description (Optional)**” cell and type-in your comment. You can do these three actions in any order.
- **Right-click** anywhere within **any** current **field row** and click on “**Insert Rows**” from the pop-up menu. This will insert a **blank row** immediately above the one where you right-clicked. You can now configure this blank row as indicated in the first bullet point above.
- **Select a range of field rows** (click *B.6.1.4*), right-click anywhere within **any** of

---

<sup>17</sup> The “**Ctrl**” key is the so-called “**Control**” key. It is usually labeled “**Ctrl**” in keyboards.

them and click on “**Insert Rows**” from the pop-up menu. This will **insert** as many **blank rows** as you had selected. These **blank rows** are inserted right above the selected range of field rows. You can now configure **each** of these blank rows as indicated in the first bullet point above.

- **Copy** and **paste** (click *B.6.1.6*) one (**or more**) existing field row(s), and then **modify** its/their field name(s), field type(s) and description(s). Copy and paste field rows is very convenient when you want to add several fields with similar configuration (same field type-size, **Required=Yes**, “**Lookup**” properties, etc.).

To decide what **field type** you **choose**, and also to possibly configure other field **properties**, you may click:

- “*D.4 How do I configure a Table field data type and size?”*”
- “*D.5 How do I configure a Table field validation rule, indexing, and other properties?”*”
- “*D.6 How do I configure the Primary Key field(s) of a Table?”*”
- “*D.7 How do I add simple and/or composite index(es) to a Table?”*”
- “*D.11.1 How do I configure a drop-down menu to enter data in a Table field?”*”

In case you leave **blank rows** between configured field rows, the blank rows will be **removed** when saving and closing the Table design.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **B.6.1.6 How do I copy/cut and paste Table fields in “Design View”?**

If the Table is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

You can then **copy** or **cut** field row(s) in the following two ways:

- Right-click anywhere within the field row and click on “**Copy**” or “**Cut**” (respectively) from the pop-up menu.
- Select one field row, or a range of consecutive field rows (click *B.6.1.4*), and then either:
  - Press “**Ctrl-c**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**c**” key) to **copy**.
  - Press “**Ctrl-x**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**x**” key) to **cut**.
  - Right-click anywhere within the selected field row(s) and click on “**Copy**” or “**Cut**” (respectively) from the pop-up menu.

You can **paste** the **previously copied** or **cut** field row(s) in either of the following ways:

- Right-click anywhere within a field row and click on “**Paste**” from the pop-up menu.
- Click anywhere within a field row, and then press “**Ctrl-v**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**v**” key).

In these two bullet points, the **copied/cut** field rows (one or more) is/are **inserted** right above the field row where you right-clicked or clicked.

- Select one field row, or a range of consecutive field rows (click *B.6.1.4*), right-click on one of the selected rows and click on “**Paste**” from the pop-up menu.
- Select one field row, or a range of consecutive field rows (click *B.6.1.4*), and then press “**Ctrl-v**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**v**” key).

In these two bullet points, the **selected** field row(s) will be **removed** (you get a box where you can confirm or cancel) and the **copied/cut** field rows (one or more) is/are **inserted** instead of the removed one(s).

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **B.6.1.7 How do I reconfigure my Table fields in “Design View”?**

If the Table is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

The **top sub-pane** shows the Table **fields** (one in each row). For each Table field name (i.e., each **row**), it contains (left to right) the **name of the field**, the **data type of the field**, and the optional **description** (free text) of the field.

Clicking on each row, you may **edit** the field’s name, **change** the field’s data type and/or **edit** the field’s description. To change the field’s data type (and size) you may click:

- “*D.4 How do I configure a Table field data type and size?”*”

Clicking on each **field** in the **top sub-pane** you will see **its** properties in the **bottom sub-pane**. You may click *B.6.1.2* for a brief description of the “**General**” properties and *B.6.1.3* for a brief description of the “**Lookup**” properties.

If you want to reconfigure the properties of your Table fields, you may click:

- “*D.5 How do I configure a Table field validation rule, indexing, and other properties?”*”
- “*D.6 How do I configure the Primary Key field(s) of a Table?”*”
- “*D.7 How do I add simple and/or composite index(es) to a Table?*”
- “*D.11.1 How do I configure a drop-down menu to enter data in a Table field?”*”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **B.6.1.8 How do I delete Table fields in “Design View”?**

If the Table is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

You can **delete** field rows in either of the following ways:

- Right-click anywhere within a field row and click on “**Delete Rows**” from the pop-up menu.

- Select one field row, or a range of consecutive field rows (click *B.6.1.4*), and then either:
  - Press the “**Supr**” key.
  - Right-click anywhere within the selected field row(s) and click on “**Delete Rows**” from the pop-up menu.

Either way you will get a box where you can **confirm** or **cancel** the **delete** operation.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **B.6.1.9 How do I reorder Table fields in “Design View”?**

If the Table is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

Select one field row, or a range of consecutive field rows (click *B.6.1.4*), that you want to relocate.

Do drag-and-drop (up or down) over any of the dark gray boxes placed to the left of the field name of the **selected** field row(s). When you press down the mouse button inside any of the dark gray boxes placed to the left of the field name of the **selected** field row(s), a thick black horizontal line will be shown. This black line shows where the selected fields will be moved to when you release the mouse button. When you move the mouse up or down (without having released the mouse button), the thick black horizontal line will move up or down, indicating where the selected fields will be moved.

You can also change the Table field order by doing **cut and paste** of field rows (click *B.6.1.6*), but I consider this **extremely risky**, and I advise you **avoid doing this**.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **B.6.2 How do I manage Table indexes in “Design View”?**


You may click:

- “*B.6.2.1 How do I view Table indexes in “Design View”?*”
- “*B.6.2.2 How do I select fields in Table indexes in “Design View”?*”
- “*B.6.2.3 How do I delete, insert or move fields in Table indexes in “Design View”?*”

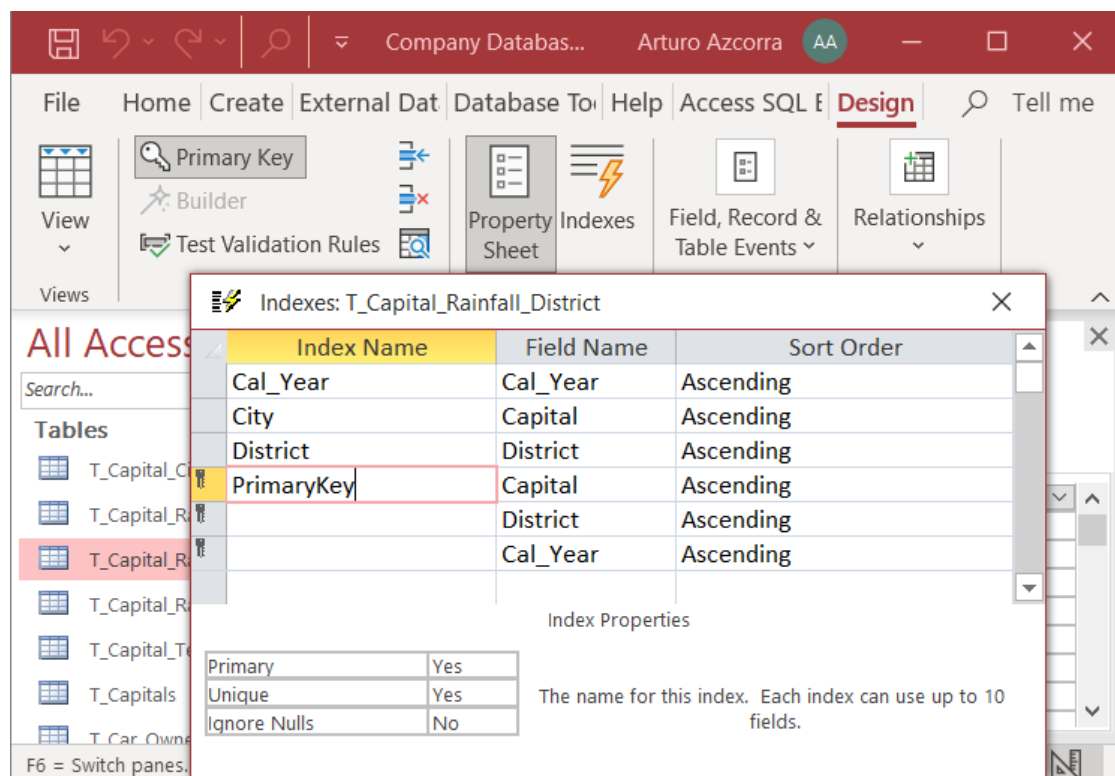
#### **B.6.2.1 How do I view Table indexes in “Design View”?**

If you rather want to **configure simple indexes**, you may click *D.5.1.8*. If you want to **configure composite or simple indexes**, you may click *D.7.2*.

To view the Table indexes, open the Table in “**Design View**” (click *B.4.1.3*).


Click on “**Design**” from the “**Ribbon-bar**” (below the “**Table Tools**” contextual label) and then click on the **Indexes**  icon. This opens a dialog box called “**Indexes**” that shows **all** the indexes in the Table and allows to **create new** indexes and/or to **configure existing** ones.

The next screenshot shows the “**Indexes**” dialog box. Notice that the heading of the dialog box has the title “**Indexes:**” followed by the Table **name** (in this case, T\_Capital\_Rainfall\_District) whose indexes are being configured:



The leftmost column (with heading “**Index Name**”) shows the **indexes** configured in this Table. Each **simple** or **composite index** is shown indicating its index name. These index names are **not** used externally, and they are **only** used within the “**Indexes**” sub-window to tell apart the different indexes defined in this Table. The index of the **Primary Key** is shown with name “**PrimaryKey**”.

The central column (with heading “**Field Name**”) shows the name(s) of the **Table fields belonging** to each index. A **simple index** does **not have**, below its **first index field row**, **another index field row** with a **blank** index name. A **composite index has**, below its **first index field row**, **another index field row** with a **blank** index name.



If an index is the one corresponding to the Table’s **Primary Key**, its “**Primary**” property will be “**Yes**”, there will be a key “” icon to the left of **all its index field rows**, and its name will usually be “**PrimaryKey**”.


In the rightmost column (with heading “**Sort Order**”) you may see the sort order of each field: “**Ascending**” or “**Descending**”.

If you configured **simple** indexing on some individual fields (click *D.5.1.8*) and/or **composite** indexes (click *D.7.2*) and/or a **simple** or **composite Primary Key** (*D.6*), **all** these indexes will be **shown** in the “**Indexes**” dialog box as follows:

- **Each simple index** is shown as one index having only one field (i.e., one row) with the index name equal to its field name (unless you had changed it). Since this is **not** the index of the **primary Key**, the property “**Primary**” of this index will be set to “**No**”. The property “**Unique**” will be set to “**Yes**” if this field was configured as

indexed without duplicate values, and to “**No**” if it was configured as indexed with duplicate values. The property “**Ignore Nulls**” will be set to “**No**”, because this is its default value (unless you changed it).

- **A simple Primary Key** (i.e., **one Key** field) will appear as one index having only one field (i.e., one row) with the index name “**PrimaryKey**” and showing the key “” icon to the left of the index name. The corresponding field name will appear to the right of the “**PrimaryKey**” index name. Since this is the index of the Table’s **Primary Key**, the properties “**Primary**” and “**Unique**” will be both set to “**Yes**”. The value of the property “**Ignore Nulls**” is irrelevant because **primary Key** fields cannot contain **Null**.
- **Each composite index** will appear as one index having several fields (i.e., several rows) with the index name that you assigned to it. One of the index field names will appear to the right of the index name, and the other fields in the composite index will appear below it (with a blank cell to their left). Since this is **not** the index of the **primary Key**, the property “**Primary**” of this index will be set to “**No**”. The property “**Unique**” will be set to “**Yes**” if this field was configured as indexed without duplicate values, and to “**No**” if it was configured as indexed with duplicate values. The property “**Ignore Nulls**” will be set to “**No**”, because this is its default value (unless you changed it).
- **A composite Primary Key** (i.e., **several Key** fields) will appear as one index with the index name “**PrimaryKey**” showing the key “” icon to the left of **all** its index field rows. One of the index field names will appear to the right of the “**PrimaryKey**” index name, and the other fields in the composite index will appear below it (with a blank cell to their left). Since this is the index of the Table’s **Primary Key**, the properties “**Primary**” and “**Unique**” will be both set to “**Yes**”. The value of the property “**Ignore Nulls**” is irrelevant because **Key** fields cannot contain **Null**.

In the screenshot above, you may see that the Table has a **composite Primary Key**, with three fields (“Capital”, “District” and “Cal\_Year”). The **composite Primary Key** is shown as a **composite index** named “**PrimaryKey**”, with its “**Primary**” property set to “**Yes**” and each of its index field rows has the key “” icon to its left. In addition to this index, there are three **simple** indexes, with **duplicate** values, each associated to the fields “Capital”, “District” and “Cal\_Year”.

Once you are done viewing your index(es), **close** the “**Indexes**” dialog box by clicking on the close “**X**” icon on its top-right corner.

You may now **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **B.6.2.2 How do I select fields in Table indexes in “Design View”?**

Open the “**Indexes**” dialog box (click *B.6.2.1*).

You **select** one **index field row** by clicking on the **small cell** on the **leftmost** side of the field row you want to select. The small cell is now highlighted in **dark gray**, and the whole **index field row** is highlighted with a **pink border**. This highlighting shows that the row has been **selected**.

You **select** a **range of contiguous index field rows** in either of the following ways:

- Click-and-drag on the **small cell** on the **leftmost** side of the **index field row** on **one of the ends** of the **range**. Click-and-drag means press the mouse button and then move the mouse up or down while keeping the mouse button pressed. While you move the mouse, you will see that the corresponding **range** of fields is highlighted: **all** the small cells on the left side of the **range** become **dark gray**, and the **whole range** of index field rows has a pink border.
- **Select** the index field row (see slightly above) on one end of the **range** that you want, and then **Shift-select** the index field row on the other end of the **range**. Doing a **Shift-select** is the same as doing a **select**, but holding the “**Shift**” key pressed while you do the mouse click.

Regardless of how you did it, you should see that the corresponding **range** of fields is highlighted: **all** the small cells on the left side of the **range** are **dark gray**, and the **whole range** of index field rows shows a **pink** border. This highlighting shows that the **range of index field rows** has been **selected**.

Notice that you **cannot** select **non-contiguous index field rows**. Also, you **cannot** use “**Ctrl+click**” to select/unselect additional **individual index field rows**.

You may now insert, delete or move (click *B.6.2.3*) the selected fields in order to reconfigure your indexes (click *D.7.3*).

### **B.6.2.3 How do I delete, insert or move fields in Table indexes in “Design View”?**

You **delete** **index field rows** in either of the following ways:

- Right-click **anywhere** inside an **index field row** and click on “**Delete Rows**” from the pop-up menu.
- **Select** a **range of index field rows** (click *B.6.2.2*) and press the “**Supr**” key. This will delete the whole **range of index field rows**.

You **insert one** **index field row** by right-clicking **anywhere** inside an **index field row** and clicking on “**Insert Rows**” from the pop-up menu. This will insert one **blank index field row right above** of the row where you right-clicked.

You **move** an **index field row**, or a **range** of them, by first **selecting** (click *B.6.2.2*) it/them and then doing drag-and-drop clicking on any **small cell** on the **left** side of the range.

If you want to reconfigure your indexes, you may click *D.7.3*.

### **B.6.3 How do I unhide/hide a Table “Property Sheet” in “Design View”?**

If the Table is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

You **unhide** the “**Property Sheet**” of a Table in either of the following ways:


- Right-click anywhere on the “Table pane” and click on “**Properties**” from the pop-up menu.
- Click on “**Design**” from the “Ribbon-bar” (below the “**Table Tools**” contextual label)



and then on the **Property Sheet** “” icon.

Either way the Table’s “**Property Sheet**” will be **unhidden**, becoming visible on the right side of the “Object Area”.


You **hide** the Table’s “**Property Sheet**” in either of the following ways:

- Click on the **close** icon “**X**” on the top-right corner of the “**Property Sheet**”.
- Right-click anywhere on the “Table pane” (except inside the “**Property Sheet**”) and click on “**Properties**” from the pop-up menu.
- Click on “**Design**” from the “Ribbon-bar” (below the “**Table Tools**” contextual label) and then on the **Property Sheet** “” icon.

Notice that the way to **hide** or **unhide** the “**Property Sheet**” is **exactly the same** (so called “toggle mode”), except for the case of hiding it by clicking on its **close** “**X**” icon.

If you want to know how to configure the “**Property Sheet**” of a Table in “**Design View**”, you may click *B.6.4*.

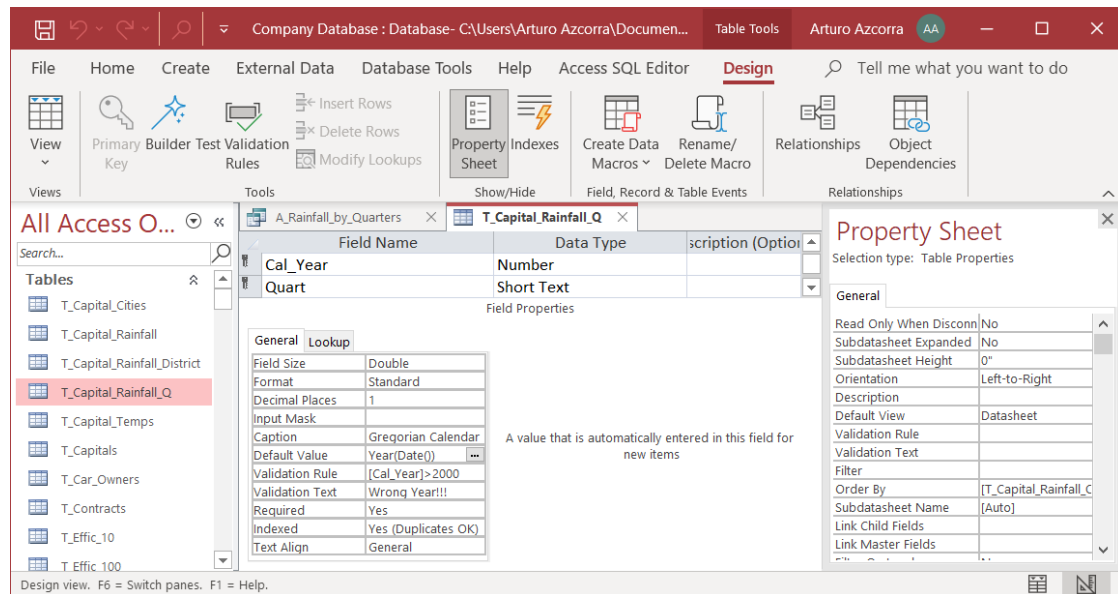
**Do not mistake** the “**Navigation Pane**” **properties** (click *B.4.1.2*) of any given object with the “**Properties Sheet**” of Tables (click *B.6.3*), Queries (click *B.7.1*) and Forms (click *B.8.1*).

As a final remark, if you **select** any object **name** (click *B.4.1.10*) in the “**Navigation Pane**”, and then you click on the **Property Sheet** “” icon from the “**Design**” Ribbon (see above), this will **show** the “**Navigation Pane**” **properties** (click *B.4.1.2*) instead of un hiding/hiding the “**Property Sheet**” of the Visible Object (click *B.2.10*), as it is indicated above. This may be very puzzling if you are not aware of it. This may be an MS-Access **bug**.

#### **B.6.4 How do I configure a Table “Property Sheet” in “Design View”?**

You first show the “**Property Sheet**” of the Table (click *B.6.3*). A Table has only one “**Property Sheet**”.

The following screenshot shows a Table with its “Property Sheet” open:



The Table’s “Property Sheet” properties include many configurable elements. Most of the configurable elements are used quite unfrequently, so I will only cover here the following two ones:

- (Table) “**Validation Rule**”:  
This is a **Boolean** expression (over the **combined** field values of each and every **new record**) that **must not** return **False**. If the **Boolean** expression returns **Null**, the record validation rule is considered correct. MS-Access will **not** allow you to enter any new record into the Table that returns **False** in the record validation rule. **The record validation rule is a very useful feature you should use a lot.** If you want to know more, you may click “[D.8 How do I configure the properties of a Table?](#)”.
- (Table) “**Validation Text**”:  
This is the text that is shown to the user in case the **record validation rule** returns **False**, to inform **why** the record cannot be entered. If you leave this blank, a default error message will be shown, in which the **Boolean** expression of the record validation rule will be displayed.

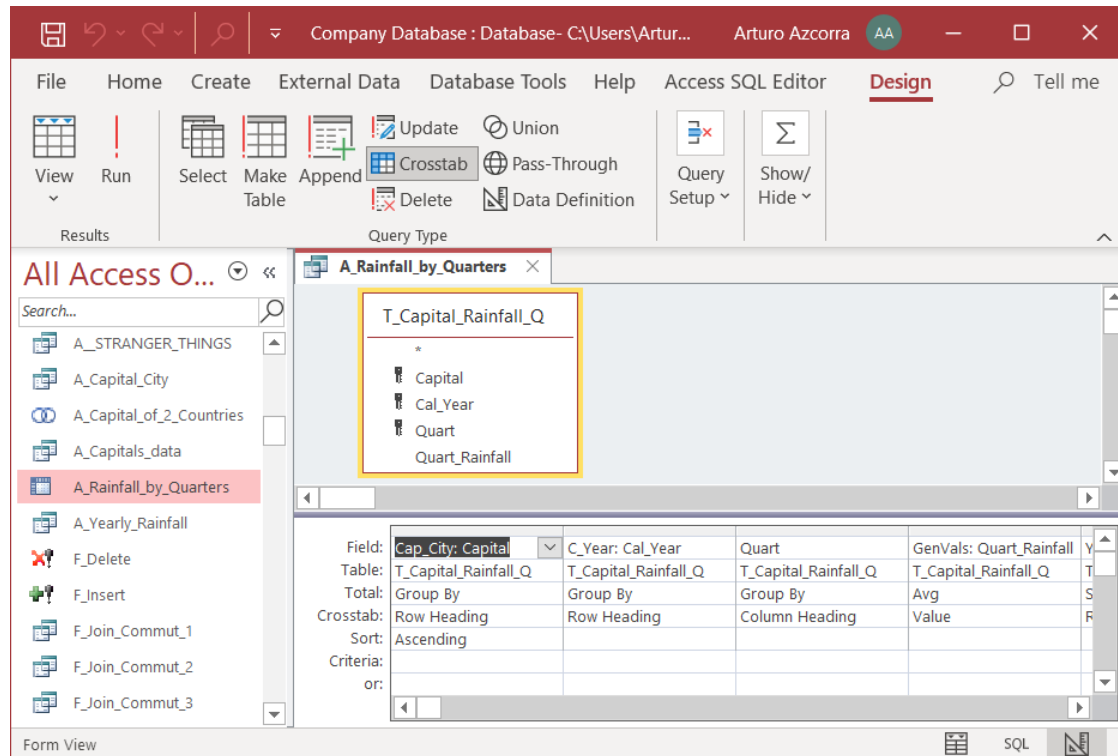
Once you are done with all your **Property** configuration, you **hide** the “Property Sheet” of the Table (click [B.6.3](#)).

## **B.7 What is a Query in “Design View”?**

A Query in “Design View” allows to **configure** the formatting and other properties of the Query results when they are shown in “**Datasheet View**”.

If you want to **create** and **configure** Queries, click [F.4.5](#).

Coming back to a “Query pane” in “**Design View**”, you will see two sub-panes, one at the top, and one at the bottom, as you may see in the next screenshot:



You can **change** the relative size of each sub-pane within the “Query pane” by doing drag-and-drop on the **line** that divides both panes. Actually, you can **hide** either sub-pane by taking the division line **all the way** to the **top**, or to the **bottom**, of the “Query pane”.

The top sub-pane shows the main **input** Table(s) or Query(es) to the Query shown in the “Query pane”. Each **input** Table or Query is shown as a **box**, with the Table/Query **name** in the heading, and its **field names** listed inside the box.

The bottom sub-pane shows the “**Query design grid**”. In this grid, each column represents **each** of the output fields of the Query, plus the “**TRANSFORM**” and “**PIVOT**” fields, plus the “**ORDER BY**” expressions (not shown in the screenshot above). The “Query design grid” is very useful to configure the formatting (i.e., the way the values will be shown) of each of the fields, by opening the Query’s “**Property Sheet**”.

If you want to configure a Query’s “**Property Sheet**”, you may click:


- “[B.7.1 How do I unhide/hide a Query’s “Property Sheet” in “Design View”?](#)”
- “[B.7.2 How do I configure a Query’s “Property Sheet” in “Design View”?](#)”

If you want to code Queries in SQL, you may click “[Part F. Writing SQL Queries to use my database](#)”.

### **B.7.1 How do I unhide/hide a Query’s “Property Sheet” in “Design View”?**


If the Query is not opened in “**Design View**”, you either **open** it (click [B.4.1.3](#)) in “**Design View**” or **change** its **view-type** (click [B.4.1.4](#)) to “**Design View**”.

You **unhide** the Query's "**Property Sheet**" in either of the following ways:

- Right-click anywhere on the "Query pane" and click on "**Properties**" from the pop-up menu.
- Click on "**Design**" from the "Ribbon-bar" (below the "**Query Tools**" contextual label) and then on the **Property Sheet**  icon.

Either way the Query's "**Property Sheet**" will be **unhidden**, becoming visible on the right side of the "Object Area".


You **hide** the Query's "**Property Sheet**" in either of the following ways:

- Click on the **close** icon "**X**" on the top-right corner of the "**Property Sheet**".
- Right-click anywhere on the "Query pane" (except inside the "**Property Sheet**") and click on "**Properties**" from the pop-up menu.
- Click on "**Design**" from the "Ribbon-bar" (below the "**Query Tools**" contextual label) and then on the **Property Sheet**  icon.

Notice that the way to **hide** or **unhide** the "**Property Sheet**" is **exactly the same** (so called "toggle mode"), except for the case of hiding it by clicking on its **close "X"** icon.

If you want to know how to configure the "**Property Sheet**" of a Query in "**Design View**", you may click *B.7.2*.

**Do not mistake** the "**Navigation Pane**" **properties** (click *B.4.1.2*) of any given object with the "**Properties Sheet**" of Tables (click *B.6.3*), Queries (click *B.7.1*) and Forms (click *B.8.1*).

As a final remark, if you **select** any object **name** (click *B.4.1.10*) in the "**Navigation Pane**", and then you click on the **Property Sheet**  icon from the "**Design**" Ribbon (see above), this will **show** the "**Navigation Pane**" **properties** (click *B.4.1.2*) instead of un hiding/hiding the "**Property Sheet**" of the Visible Object (click *B.2.10*), as it is indicated above. This may be very puzzling if you are not aware of it. This may be an MS-Access **bug**.

### **B.7.2 How do I configure a Query's "Property Sheet" in "Design View"?**

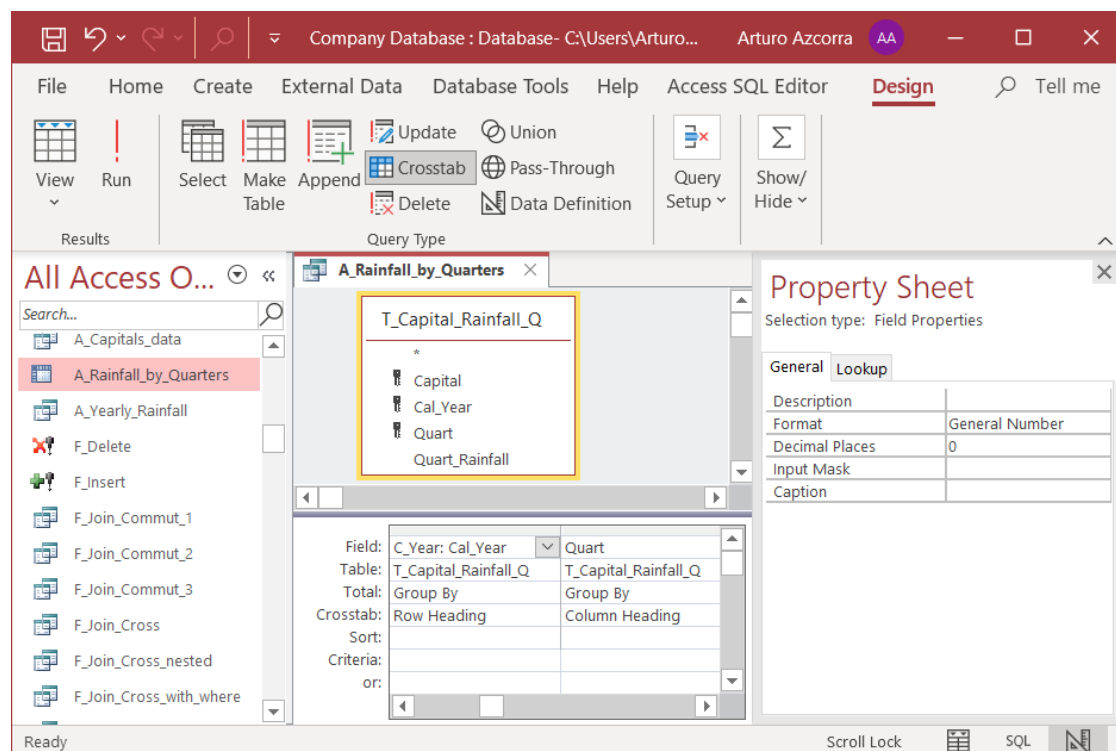
A Query has **several** "**Property Sheet**" that can be shown: one for the Query as a whole (labeled "**Query Properties**"), one for **each output field** (labeled "**Field Properties**") and one for **each input record-list** (labeled "**Field List Properties**").

You first show the Query's "**Property Sheet**" (click *B.7.1*). You then show the specific properties that you want, as follows:

- You show the "**Query Properties**" of the Query as a whole, in either of the following ways:
  - Click on the **gray** background of the **top** sub-pane.
  - Click on the **white** background of the **bottom** sub-pane.
  - Click anywhere **inside** a **column** (in the **bottom** sub-pane) that **does not** correspond to **any output field**.

- You show the “**Field Properties**” of a given **output field** by clicking anywhere **inside the column** (in the **bottom** sub-pane) corresponding to **that output field**.
- You show the “**Field List Properties**” of a given **input field value-list**, by clicking anywhere on the **box** (in the **top** sub-pane) representing that **input field value-list**.

The following screenshot shows the “**Property Sheet**” of the Query “A\_Rainfall\_by\_Quarters” corresponding to the “**Field properties**” of its **output field** “Cal\_Year”:



The “**Query properties**” shown under the “**General**” tab include many configurable elements, but most of them are used very unfrequently. I will therefore only cover the “**RecordsetType**” property. I advise you always configure it to “**Snapshot**” to prevent Table modification from Queries (click *K.4.4*).

The “**Field Properties**” shown for **each output field** under the “**General**” tab are the following:

- “**Description**”: allows to add a comment to this field.
- “**Format**”: allows to set the **format** in which values are presented (click *H.6*). **This is a very useful property**. You can configure the format you want for the values in this field by typing-in the format value you want or by clicking on the rightmost side of the “**Format**” row and clicking on an option from the drop-down menu.
- “**Decimal Places**”: the number of decimals to be shown (**only** for **numeric** fields).
- “**Input mask**”: allows to restrict input values using masks (click *D.5.2.4*).
- “**Caption**”: allows to show in “**Datasheet View**” the name you write here as column header, instead of the actual output field name.
- “**Text Format**”: allows to select between “**Plain Text**” and “**Rich Text**” (only for

**Short Text** and **Long Text** fields).

The “**Field List Properties**” shown for each **input field value-list** under the “**General**” tab are “**Alias**” and “**Source**”. They are used very unfrequently, so I will not cover them here.

Finally, you hide the “**Property Sheet**” of the Query (click *B.7.1*).

## **B.8 What is a Form in “Design View”?**

A Form in “**Design View**” allows to **configure** the formatting of fields, the supporting Table and other relevant properties of the Form.

If you want to **create** and **configure** Forms, click *D.10*.



The “**Property Sheet**” of Tables and Queries can only be unhidden and configured in “**Design View**”. However, in the case of Forms, the “**Property Sheet**” can be unhidden and configured in either “**Datasheet View**” or “**Design View**”. If you want to configure a Form’s “**Property Sheet**”, you may click:

- “*B.8.1 How do I unhide/hide a Form’s “Property Sheet” in “Datasheet View” or “Design View”?*”
- “*B.8.2 How do I configure a Form’s “Property Sheet” in “Datasheet View” or “Design View”?*”

### **B.8.1 How do I unhide/hide a Form’s “Property Sheet” in “Datasheet View” or “Design View”?**

If the Form is not opened in “**Datasheet View**” or “**Design View**”, you either **open** it (click *B.4.1.3*) or **change** its **view-type** (click *B.4.1.4*). The Form should now be opened in either “**Datasheet View**” or “**Design View**”.



You **unhide** the Form’s “**Property Sheet**” in either of the following ways:

- Right-click anywhere on the “Form pane” and click on “**Properties**” from the pop-up menu.
- If the Form is opened in “**Datasheet View**”, click on “**Datasheet**” from the “Ribbon-bar” (below the “**Form Tools**” contextual label) and then on the **Property Sheet** “” icon.
- If the Form is opened in “**Design View**”, click on “**Design**” from the “Ribbon-bar” (below the “**Form Design Tools**” contextual label) and then on the **Property Sheet** “” icon.

Either way the Form’s “**Property Sheet**” will be **unhidden**, becoming visible on the right side of the “Object Area”.

You **hide** the Form’s “**Property Sheet**” in either of the following ways:


- Click on the **close** icon “**X**” on the top-right corner of the “**Property Sheet**”.
- Right-click anywhere on the “Form pane” (except inside the “**Property Sheet**”) and click on “**Properties**” from the pop-up menu.

- If the Form is opened in “**Datasheet View**”, click on “**Datasheet**” from the “**Ribbon-bar**” (below the “**Form Tools**” contextual label) and then on the **Property Sheet** “” icon.
- If the Form is opened in “**Design View**”, click on “**Design**” from the “**Ribbon-bar**” (below the “**Form Design Tools**” contextual label) and then on the **Property Sheet** “” icon.

Notice that the way to **hide** or **unhide** the “**Property Sheet**” is **exactly the same** (so called “toggle mode”), except for the case of hiding it by clicking on its **close “X”** icon.

If you want to know how to configure the “**Property Sheet**” of a Form you may click *B.8.2*.

**Do not mistake** the “**Navigation Pane**” **properties** (click *B.4.1.2*) of any given object with the “**Properties Sheet**” of Tables (click *B.6.3*), Queries (click *B.7.1*) and Forms (click *B.8.1*).

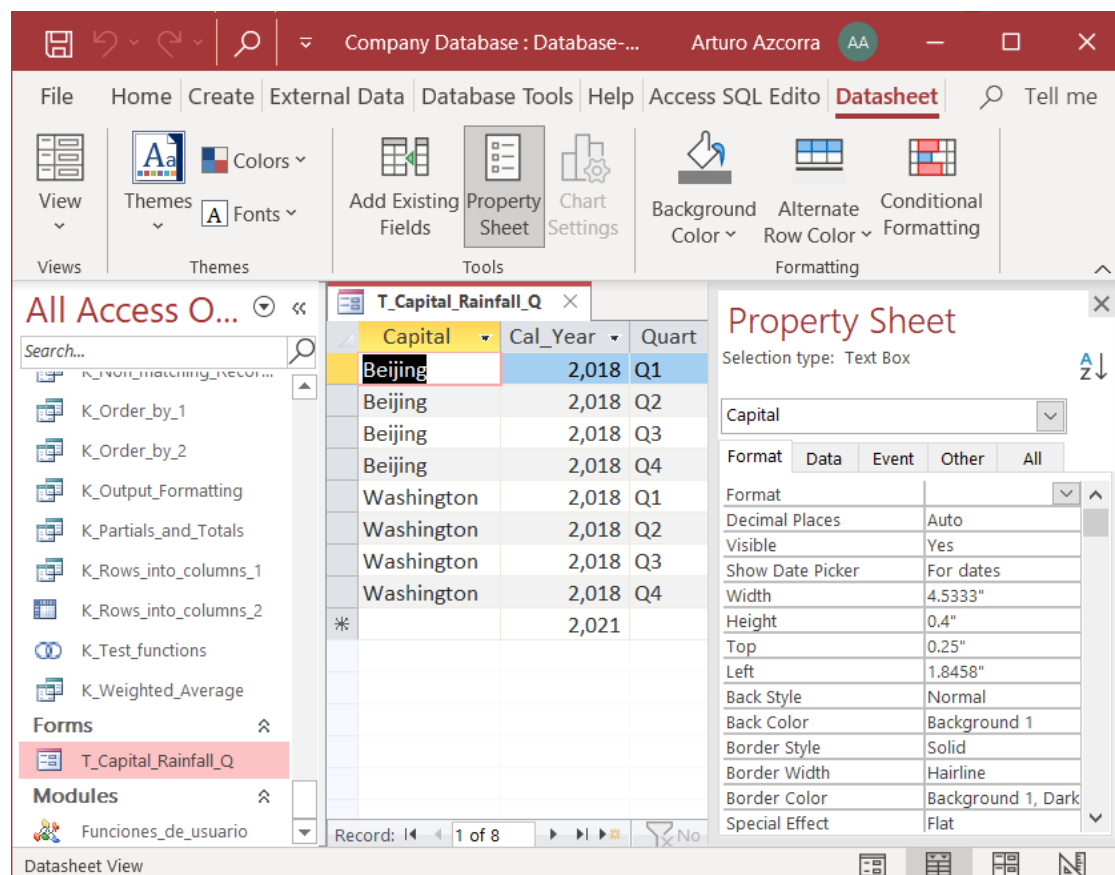
As a final remark, if you **select** any object **name** (click *B.4.1.10*) in the “**Navigation Pane**”, and then you click on the **Property Sheet** “” icon from the “**Design**” Ribbon (see above), this will **show** the “**Navigation Pane**” **properties** (click *B.4.1.2*) instead of unhiding/hiding the “**Property Sheet**” of the Visible Object (click *B.2.10*), as it is indicated above. This may be very puzzling if you are not aware of it. This may be an MS-Access **bug**.

### **B.8.2 How do I configure a Form’s “Property Sheet” in “Datasheet View” or “Design View”?**

A Form has **several** “**Property Sheet**” that can be shown, one for **each** of the different **elements** of the Form. The name of the **element** whose properties are being shown is displayed in an **element box** with a drop-down menu that is placed at the top-left side of the “**Property Sheet**”.



In the following screenshot you can see the “**Property Sheet**” of the Form “**T\_Capital\_Rainfall\_Q**” (that is open in “**Datasheet View**”), showing the properties of the field “**Capital**”:



You first **unhide** the Form’s “**Property Sheet**” (click *B.8.1*). You can then **select** the **element** whose **properties** are being **shown** in the following two ways:

- Click on the element you want in the “Form pane” (regardless of it being in “**Datasheet View**” or “**Design View**”).
- Click on the drop-down menu “**☐**” icon of the **element box** at the top-left side of the “**Property Sheet**” and select the **element** you want by clicking on its name from the drop-down menu.

Either way, you will see that the **name** of the **element** that you clicked is displayed in the **element box** at the top-left side of the “**Property Sheet**”, and its corresponding properties are shown.

The Form’s **elements** are grouped in four “**Selection types**”. When you **select** a **given element** so its properties are shown, the **element’s** “**Selection type**” will be displayed at the top of the “**Property Sheet**”. The four “**Selection types**” are:


- “**Form**”:  
Shows the properties of the **Form** as a whole.
- “**Label**”:  
Shows the properties of one of the Form’s **column headings**. The specific column heading whose properties are shown is indicated in the element box by the column

name followed by “\_Label”.

- **“Text Box”**: Shows the properties of one of the Form’s **columns**, whose name is indicated in the element box.
- **“Section”**: Shows the properties of the **“FormFooter”**, **“FormHeader”** or **“Detail”**, as shown in the **element** box.

The properties of the **selected element** (indicated in the element box) are shown in **five** tabs:

- **“Format”**: Shows the **value formatting** properties of the element.
- **“Data”**: Shows properties related to the **management of data** in the element.
- **“Event”**: Shows properties related to each **“event”** (i.e., click, open, close, ...) you can do on the element. The properties allow to invoke **user-defined VBA subroutines** or associate other actions to these events. **The “Event” properties make Forms very powerful and flexible.**
- **“Other”**: Shows **other** properties of the element.
- **“All”**: Shows **all** the properties of the **element**, this is, all the properties listed in the other four tabs.

Since Forms have **many** properties, in order to help you find the one you want, you can choose between showing them in alphabetical order or in their default order. To toggle between both ordering types, click on the **A-Z**  icon placed at the top right corner of the **“Property Sheet”**.

Finally, you hide the **“Property Sheet”** of the Form (click *B.8.I*).

## **B.9 What is a Query in “SQL View”?**

A Query in **“SQL View”** allows to **view** and **edit** the SQL code of the Query.

If you want to **create** and **configure** Queries, click *F.4.5*.

As of the writing of this book, the editing capacities in **“SQL View”** are extremely poor. Microsoft announced that it would add the Monaco SQL editor to MS-Access along the second Quarter of 2021.

Unless the Monaco editor is finally incorporated to MS-Access, my advice is that you **never** use **“SQL View”**, and you rather edit your SQL code using the plug-in **“Access SQL Editor”**. If you want to know more about this, you may click *“F.5 How do I edit my SQL Queries with the plug-in “Access SQL Editor”?”*.

Just for the record, the **“SQL View”** has a specific **“Property sheet”**.

If you want to code Queries in SQL, you may click “*Part F. Writing SQL Queries to use my database*”.

## **B.10 What is the “Relationships” pane?**

It is a **tool** to **create**, **view**, **modify** and **delete** the Relationships (click *C.11*) that you want to establish between your Tables.




If you want to **create** and **configure** Relationships, you may click “*D.9 How do I create and configure my Table Relationships?*”.

If you want to know more about the “**Relationships**” pane, you may click:

- “*B.10.1 How do I open the “Relationships” pane?*”
- “*B.10.2 How do I view my Relationships?*”
- “*B.10.3 How do I view a Relationship’s properties?*”
- “*B.10.4 How do I change the layout of the “Relationships” pane?*”
- “*B.10.5 How do I edit an existing Relationship?*”
- “*B.10.6 How do I create a new Relationship?*”
- “*B.10.7 How do I delete a Relationship?*”
- “*B.10.8 How do I close the “Relationships” pane?*”

### **B.10.1 How do I open the “Relationships” pane?**

You open the “**Relationships**” pane in either of the following ways:

- Click on “**Database Tools**” from the “Ribbon-bar”, and then on the **Relationships**  icon.
- If the currently viewed object is a **Table** in “**Datasheet View**”, click on “**Table**” from the “Ribbon-bar” (below the “**Table Tools**” contextual label), and then on the **Relationships**  icon.
- If the currently viewed object is a **Table** in “**Design View**”, click on “**Design**” from the “Ribbon-bar” (below the “**Table Tools**” contextual label), and then on the **Relationships**  icon.

Either way will open the “**Relationships**” pane in the “Object Area” (click *B.2.8*) of the MS-Access window.

### **B.10.2 How do I view my Relationships?**

The **Relationships** are viewed with the “**Relationships**” pane.

Open the “**Relationships**” pane (click *B.10.1*).


If you have not created any Relationships, the “**Relationships**” pane will be blank (gray background).

If you have already created Relationships, it may happen that the “**Relationships**” pane is **not** showing **all** your Relationships because it is possible to **hide Table-boxes**

(click *B.10.4.5*), which also hides **all** their Relationships. My advice is you **do not hide Table-boxes**, and therefore, **all** the Relationships are shown in the “**Relationships**” pane.

When you have created Relationships, **each Table** having Relationships is **represented** in the “**Relationships**” pane as one (or more) **Table-boxes**.

**Each Table-box** has:

- A **heading** with the **name** of the **Table** that the **Table-box** represents, possibly followed by the **suffix** “\_1”, “\_2”, “\_3”, etc. (click *B.10.4.4*).
- A **list** of **all** the **field names** of the **Table** that the **Table-box** represents, in the same order as they are defined in the Table.
- If the **list of field names does not fit** in the current size of the **Table-box**, a **vertical scrollbar** will be shown on the right side of the **Table-box**.
- A key “” icon placed to the **left** of **each** of the field names of the **Primary Key** of the **Table** that the **Table-box** represents.

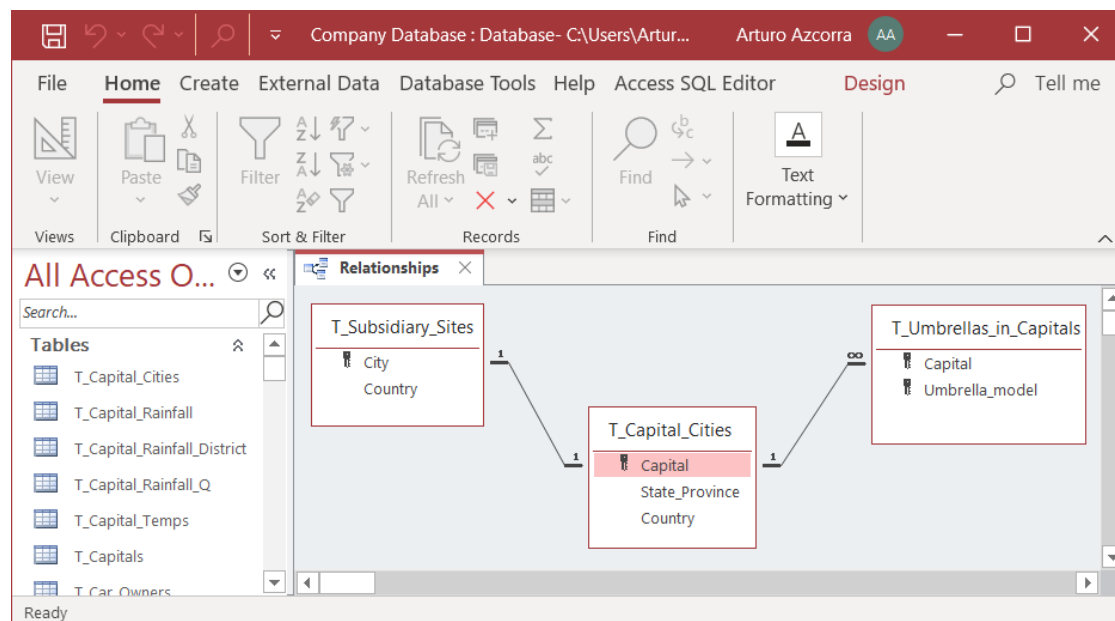
**Each Relationship** is **represented** as one (or more) **connecting lines** between the **master Table-box** and its **slave Table-box**. **Each line connects one different master-slave field pair** in the Relationship. Therefore, if the Relationship involves “**n**” **master-slave field pairs**, then the Relationship is represented as “**n**” **connecting lines** between its **master Table-box** and its **slave Table-box**, one line for each master-slave field pair.

The **crooked endpoints** of **each** of the Relationship’s **connecting lines** are labeled to identify the Relationship’s **type**, as follows:

- **One-to-many** Relationships (click *C.11.2*) **with referential integrity** (click *C.11.1*):  
**Each** of the Relationship’s **connecting lines** has a “**1**” at the crooked endpoint of the **master Table-box** and one infinity symbol “**∞**” at the crooked endpoint of the **slave Table-box**.
- **One-to-one** Relationships (click *C.11.2*) **with referential integrity** (click *C.11.1*):  
**Each** of the Relationship’s **connecting lines** has a “**1**” at **both** crooked endpoints.
- Relationships **without referential integrity** (click *C.11.1*):  
**Each** of the Relationship’s **connecting lines** has a dot “**.**” at **both** crooked endpoints.

Notice that in **one-to-one** Relationships **with** referential integrity, and in Relationships **without** referential integrity, you **cannot** visually distinguish in the “**Relationships**” pane which is the **master Table-box** and which is the **slave Table-box**.

The next screenshot shows the “**Relationships**” pane with **three** Tables and **two** Relationships **with** referential integrity: a **one-to-one** and a **one-to-many**:



As you may see in the screenshot above, **connecting lines** representing Relationships may come out of either the **left** side and/or the **right** side of the field’s name: this does not have any meaning, and it is only a layout matter.

A given **field** from a given **Table-box** will have as **many connecting lines** as the **number** of Relationships with **other Table-boxes** in which the field is a **master field**. In the screenshot above you may see that the field “Capital” from the **Table-box** “T\_Capital\_Cities” has **two** connecting lines (one on the **right** and one on the **left**), because it is a **master field** in **two** Relationships with **two other Table-boxes**.

If a given **Table field** is involved in **more than one** Relationship as a **slave field**, MS-Access will show as **many Table-boxes** for that **Table**, as the **number** of Relationships in which the **Table field** is a **slave field**. This is required to have **each Relationship** represented as a **set of connecting lines** between **two different Table-boxes**.





If the layout of **Table-boxes** and Relationships is **too large** to fit in the current viewing area of the “**Relationships**” pane, a vertical and/or horizontal scrollbar will be shown (see the screenshot above). You can use these scrollbars to select what part of the complete diagram of **Table-boxes** and Relationships is shown in the viewing area of the “**Relationships**” pane.

Once you are done viewing your Relationships, close the “**Relationships**” pane (click *B.10.8*).

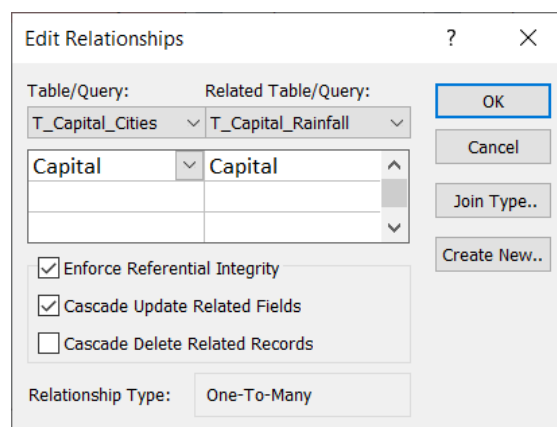
### **B.10.3 How do I view a Relationship’s properties?**

To **view** the **properties** of a **Relationship**, you have to **first view** the **Relationship** (click *B.10.2*) in the “**Relationships**” pane. You can then **view** the **properties** of the **Relationship** in either of the following ways:

- Double-click on **any of the lines** that represent the Relationship. Remind that you have to double-click on the **straight part** of the line and **not** on either of its crooked endpoints.

- Right-click on one of the **lines** that represents the Relationship and click on “**Edit Relationship**” from the pop-up menu. Remind that you have to right-click **on the straight part** of the line and **not** on either of its crooked endpoints.
- Select the Relationship by clicking on one of the **lines** that represents it (the line will become **highlighted in bold**). Remind that you have to click **on the straight part** of the line and **not** on either of its crooked endpoints. After having selected the Relationship, click on the **Edit Relationships**  icon from the “**Relationship Tools / Design**” contextual Ribbon.
- Click on the **Edit Relationships**  icon, from the “**Relationship Tools / Design**” contextual Ribbon when no Relationship is selected (i.e., when no Relationship line is highlighted in bold). This will open a **blank “Edit Relationships”** box where you will have to select the name of the two Table-boxes that identify the Relationship whose properties you want to view. You can select the name of each of the two Table-boxes clicking on the drop-down menu “” icon below the labels “**Table or Query**” (**master** Table) and “**Related Table/Query**” (**slave** Table) at the top of the “**Edit Relationships**” box. Notice that you have to select **first** the **master** Table (i.e., the Table under the label “**Table or Query**”).
- Double-click on the background of the “**Relationships**” pane, on any Table-box field or on any **crooked** line endpoint. This will open a **blank “Edit Relationships”** box where you will have to select the name of the two Table-boxes that identify the Relationship whose properties you want to view. You can select the name of each of the two Table-boxes clicking on the drop-down menu “” icon below the labels “**Table or Query**” (**master** Table) and “**Related Table/Query**” (**slave** Table) at the top of the “**Edit Relationships**” box. Notice that you have to select **first** the **master** Table (i.e., the Table under the label “**Table or Query**”).
- Drag-and-drop **any field** from **either** of the two Table-boxes of the Relationship to **any field** of **the other** Table-box. MS-Access will then show a dialog box asking if you want to **edit** the **existing** Relationship (or create a new one), where you should click on “**Yes**”.

Regardless of which of the above ways you used, you will end with an “**Edit Relationships**” box where you can **view the properties** of the Relationship you wanted. The following screenshot shows an “**Edit Relationships**” box (this corresponds to an already existing Relationship because the top-right button is “**OK**”):



Table/Query:	Related Table/Query:
T_Capital_Cities	T_Capital_Rainfall
Capital	Capital

Enforce Referential Integrity  
 Cascade Update Related Fields  
 Cascade Delete Related Records

Relationship Type: One-To-Many

The “**Edit Relationships**” box shows the following properties:

- **Master Table-box name**  
Placed on the top-left, below the label “**Table/Query:**”, with gray background. What is shown is **not** the **master Table name**, but rather the **name of one of the Table-boxes** that represent the **master Table**. Remind that the “**Relationships**” pane may show **several Table-boxes** (click *B.10.4.4*) to represent a given Table.
- **Master field names**  
Placed on the top-left, below the name of the master Table-box, with white background. Each **master field name** is in one **row**. On the **right** of each **master field** made you can see **its corresponding slave field name**.
- **Slave Table-box name**  
Placed on the top-right, below the label “**Related Table/Query:**”, with gray background. What is shown is **not** the **slave Table name**, but rather the **name of one of the Table-boxes** that represent the **slave Table**. Remind that the “**Relationships**” pane may show **several Table-boxes** (click *B.10.4.4*) to represent a given Table.
- **Slave fields**  
Placed on the top-right, below the name of the **slave Table-box**, with white background. Each **slave field name** is in one **row**. On the **left** of each **slave field name** you can see **its corresponding master field name**.
- **Relationships’ referential integrity**  
If the “**Enforce Referential Integrity**” checkbox is **ticked**, this Relationship has **referential integrity** (click *D.9.3*). If it is **unticked**, this Relationship does **not** have **referential integrity**.
- “**Cascade Update Related Fields**” checkbox  
If this checkbox is **ticked**, when you change the value of a **master field** in a **master record**, the system will **automatically update** (click *D.9.4*) the value of the **slave field**, in **all its corresponding slave records**. If it is **unticked**, the system **will not** perform the automatic update.
- “**Cascade Delete Related Records**” checkbox  
If this checkbox is **ticked**, when you **delete** a **master record**, the system will **automatically delete** (click *D.9.5*) **all its corresponding slave records**. If it is **unticked**, the system **will not allow you to remove** a **master record** until you have **previously removed all its slave records**.
- **Relationship Type**  
Placed at the bottom-center, the cell to the right of the “**Relationship Type**” label shows the **type** of this Relationship. The Relationship type can be “**One-To-Many**” (click *C.11.2*), “**One-To-One**” (click *C.11.2*) or “**Indeterminate**” (click *C.11.3*).

Once you are done viewing your Relationships, close the “**Relationships**” pane (click *B.10.8*).



## B.10.4 How do I change the layout of the “Relationships” pane?

You may click:

- “B.10.4.1 How do I select/unselect Table-boxes?”
- “B.10.4.2 How do I unhide/hide Relationships?”
- “B.10.4.3 How do I unhide/hide the “Add Tables” sub-pane?”
- “B.10.4.4 How do I unhide or add Table-boxes?”
- “B.10.4.5 How do I hide or delete a Table-box?”
- “B.10.4.6 What is the difference between unhide/hide and add/delete a Table-box?”
- “B.10.4.7 How do I move a Table-box?”
- “B.10.4.8 How do I resize a Table-box?”
- “B.10.4.9 How do I front-display a Table-box?”
- “B.10.4.10 How do I save the “Relationships” pane layout?”

### B.10.4.1 How do I select/unselect Table-boxes?

You **select** one, or more, Table-box(es) in either of the following ways:

- Click on a Table-box **frame**, on its **header**, or on any of its **fields**.
- Click-and-drag the mouse inside the “**Relationships**” pane. This will **show** a **rectangle** having one corner where you pressed the mouse-button, and the opposite corner at the current position of the mouse. All the Table-boxes that lie (totally or partially) **within** the **shown rectangle** will be **selected** when you **release** the mouse button.

When a Table-box is **selected**, its **frame** is **highlighted** in **yellow**.

You **select** one, or more, **additional** Table-box(es) in either of the following ways:

- Control-click on the Table-box **frame**, on its **header**, or on any of its **fields**.
- Control-click-and-drag the mouse inside the “**Relationships**” pane. This will **show** a **rectangle** having one corner where you pressed the mouse-button, and the opposite corner at the current position of the mouse. All the Table-boxes that lie (totally or partially) **within** the **shown rectangle** will be **additionally selected** when you release the mouse button.


You **unselect one** selected Table-box by **Control-clicking** on its **frame**, on its **header**, or on any of its **fields**.

You **unselect all** the selected Table-boxes by clicking on the gray background of the “**Relationships**” pane.


For all the indications above, if you click **between** the fields, or on the blank space **below the last field**, it will have no effect.

### B.10.4.2 How do I unhide/hide Relationships?


You **unhide** all the **Relationships** in either of the following ways:

- Right-click anywhere on the gray background of the “**Relationships**” pane and click on “**Show All Relationships**” from the pop-up menu.
- Click on “**Design**” from the “Ribbon-bar” (below the “**Relationship Tools**” contextual label), and then click on the **All Relationships**  icon.

You **unhide** all the Relationships of **one unhidden Table-box** in either of the following ways:

- Right-click on the Table-box **header**, or on any of its **fields**, and click on “**Show Direct Relationships**” from the pop-up menu.
- **Select the** Table-box (click *B.10.4.1*) and click on the **Direct Relationships**  icon from the “**Relationship Tools / Design**” contextual Ribbon. If you had selected **several** Table-boxes, this action is applied to the **last selected** Table-box.

You **unhide** one Relationship by **unhiding** its two Table-boxes (click *B.10.4.4*).

You **hide** all the **Relationships** (and all the Table-boxes) by clicking on “**Design**” from the “Ribbon-bar” (below the “**Relationship Tools**” contextual label), and then clicking on the **Clear Layout**  icon. Notice that this will also **clear** the “**Relationships**” pane layout.


You **hide** a Relationship by **hiding** one of its two Table-boxes.

Note that whenever you **unhide**/**hide** a Relationship, you **also unhide**/**hide** its **two Table-boxes** (respectively). Note that whenever you **unhide** a Table-box (click *B.10.4.4*), **all** its Relationships with other visible Table-boxes are **also unhidden**.

For all the indications above, if you click **between** the fields, or on the blank space **below the last field**, it will have no effect.

### B.10.4.3 How do I unhide/hide the “Add Tables” sub-pane?


You **unhide** the “**Add Tables**” sub-pane in either of the following ways:

- Right-click on the gray background of the “**Relationships**” pane and click on “**Show Table...**” from the pop-up menu.
- Click on “**Design**” from the “Ribbon-bar” (below the “**Relationships Tools**” contextual label), and then click on the **Add Tables**  icon.

Either way will show the “**Add Tables**” sub-pane on the right side of the “**Relationships**” pane. The “**Add Tables**” sub-pane shows (clicking on the “**Tables**” tab at its top) a **list** of **all** the (non-system) **local** Tables. Notice that **linked** Tables are listed by clicking on the “**Links**” tab at its top.

You **select** Table **names** from the “**Add Tables**” sub-pane by clicking on one Table **name** and possibly doing **Shift+click** and/or **Ctrl+click** on other Table **names**.

You **unhide** the “**Add Tables**” sub-pane in either of the following ways:

- Click on the close icon “**X**” on its top-right corner
- Click on “**Design**” from the “**Ribbon-bar**” (below the “**Relationships Tools**” contextual label), and then click on the **Add Tables** “” icon.

#### **B.10.4.4 How do I unhide or add Table-boxes?**

The procedure to **unhide** or **add** (click *B.10.4.6*) a Table-box is **exactly the same**.

You **unhide** or **add** one, or more, **Table-boxes** (each of them representing a **different Table**) in either of the following ways:

- Drag-and-drop the **Table name** from the “**Navigation Pane**” to the “**Relationships**” pane.
- Double-click on a **Table name** in the “**Add Tables**” sub-pane (click *B.10.4.3*).
- Drag-and-drop a **Table name** from the “**Add Tables**” sub-pane (click *B.10.4.3*) to the “**Relationships**” pane.
- **Select** any number of **Table names** on the “**Add Tables**” sub-pane (click *B.10.4.3*) and click on the “**Add Selected Tables**” button (at the bottom of the “**Add Tables**” sub-pane). This will **unhide** or **add one Table-box** representing **each** of the selected Table names.

Regardless how which way you chose, if the Table **name** has one, or more, hidden Table-box(es) representing it, then the **hidden Table-box with lowest integer suffix**<sup>18</sup> (see below) will be **unhidden**. Otherwise, a **new Table-box** named with the corresponding integer suffix (see below) will be **added**.

Note that whenever you **unhide** a Table-box, **all** its Relationships with **other visible Table-boxes** are **also unhidden**. Note that whenever you **unhide** a Relationship (click *B.10.4.2*), its **two** Table-boxes are **also unhidden**.

The **Table-box name** of a **newly added Table-box** representing a given **Table** is determined as follows:

1. If there is no existing Table-box (**hidden** or **unhidden**) with the same **name** as the **Table name**, then, the **Table-box name** will be the **Table name**.
2. Otherwise, the **Table-box name** will be the **Table name** with the suffix “**\_n**” where “**n**” is the **lowest integer** such that there is **no other Table-box** (hidden or unhidden) with that same name.

Because of this naming algorithm, my advice is you **never** name your Tables using the suffix “**\_1**”, “**\_2**”, “**\_3**”, etc. If you really need to name a series of Tables with sequential suffixes, use “**\_a**”, “**\_b**”, “**\_c**”, etc. (click *D.2.4.6*).


#### **B.10.4.5 How do I hide or delete a Table-box?**

The procedure to **hide** or **delete** (click *B.10.4.6*) a Table-box is **exactly the same**.

You **hide** or **delete** a Table-box in either of the following ways:

---

<sup>18</sup> For this purpose, having no suffix is considered lower than suffix number 1.

- Right-click on the Table-box **header**, or on any of its **fields**, and click on “**Hide Table**” from the pop-up menu.
- **Select** the Table-box(es) that you want (click *B.10.4.1*) and then click on the **Hide Table** “” icon from the “**Relationship Tools / Design**” contextual Ribbon.

When doing the above, if the **Table-box** has **at least one** Relationship, then the Table-box is **hidden**. If the Table-box rather has **no** Relationships, then the Table-box is **deleted**. Remind that there can exist Relationships that are **not** currently **shown**, because one of their two Table-boxes is **hidden**.

When a Table-box is **hidden**, **all** its Relationships are **also hidden**.

If a Table has **several unhidden** Table-boxes, and you want to **hide** or **delete all** of them, you have to **manually hide** or **delete each** of them.

If you **hide** or **delete** Table-boxes, the **layout** that you had configured for them (i.e., their **size** and **position** within the “**Relationships**” pane) **is lost**. Therefore, when they are **unhidden** or **added**, MS-Access will decide in what **size** and **position** they are depicted, regardless of how you had previously arranged them.

For all the indications above, if you click **between** the fields, or on the blank space **below the last field**, it will have no effect.

#### **B.10.4.6 What is the difference between unhide/hide and add/delete a Table-box?**

**Hiding** a Table-box means that it becomes **not** visible, **but** it will **become visible** when you **unhide** all the Relationships and their Table-boxes (click *B.10.4.2*).

**Deleting** a Table-box means that it becomes **not** visible, and it will **not become visible** when you **unhide** all the Relationships and their Table-boxes (click *B.10.4.2*). Actually, the Table-box has been deleted (not hidden), and its **integer suffix** (click *B.10.4.4*) has been cleared.

When you **hide/delete** (click *B.10.4.5*) a Table-box that **is involved in at least one Relationship**, the Table-box will actually be **hidden**.

When you **hide/delete** (click *B.10.4.5*) a Table-box that is **not** involved in **any Relationship**, the Table-box will actually be **deleted**. If you later need this Table-box, you will have to **add** a new Table-box (click *B.10.4.4*) for that Table.

#### **B.10.4.7 How do I move a Table-box?**

You **move a Table-box** by doing drag-and-drop on its **heading**. I strongly recommend you **move Table-boxes** to place them in a clear and useful layout, so you may **properly see all the Table-boxes and all their Relationships**. You would typically place the **master** Table-boxes with many Relationships in the middle of the “**Relationships**” pane, and Table-boxes with less Relationships around them. It is also important to place all Table-box replicas one on top of another, and with the same size, to visualize that all of them represent **the same Table**.

Do not forget to **save** the “**Relationships**” pane **layout** after your changes (click *B.10.4.10*).

#### **B.10.4.8 How do I resize a Table-box?**

**Click-and-drag** on one of its frame **corners** or frame **sides**. When you **press** the mouse button **while** the mouse cursor is on the frame of the Table-box the mouse icon changes to the **resize** icon. You can then **drag** the mouse to enlarge or reduce the Table-box. Once you **release** the mouse button, the Table-box will **stay** with the **size** you configured.

You may also automatically resize a Table-box to adjust its **width** its longest field name, and its **height** to its number of fields. To do this, right-click on the Table-box header, or on any of its field names, and click on “**Size to Fit**” from the pop-up menu. Notice however that the resizing does not take into account the Table-box name, so its width may conceal part of it: this on my view severely limits the usefulness of this feature.

For all the indications above, if you click **between** the fields, or on the blank space **below the last field**, it will have no effect.

#### **B.10.4.9 How do I front-display a Table-box?**

You **front-display** a **Table-box** by just **selecting** it (click *B.10.4.1*).

**Front-displaying** a Table-box means displaying it **over** the **other** Table-boxes with which it overlaps. **Front-displaying** Table-boxes is **very useful** to configure the Relationship’s **layout** that you want.

#### **B.10.4.10 How do I save the “Relationships” pane layout?**

You save the **layout** of the “**Relationships**” pane in either of the following ways:

- Right-click on the **tab** of the “**Relationships**” pane and click on “**Save**” from the pop-up menu.
- Right-click anywhere on the gray background of the “**Relationships**” pane and click on “**Save Layout**” from the pop-up menu.

Once you have **saved** your “**Relationships**” pane **layout**, you may **close** it (click *B.10.8*).

Notice that you **do not** need to **save** the changes on the **Relationships themselves**: if you **create**, **modify** or **delete** Relationships, these changes are **automatically saved**. You only need to **save** the **layout** of the “**Relationships**” pane.

You can also save the **layout** changes of the “**Relationships**” pane by **closing** it (click *B.10.8*) and in the dialog box that will pop-up with the warning message “*Do you want to save changes to the layout of 'Relationships'?*” you click on “**Yes**”. Although you can save the layout in this way, it is **risky**, and it is therefore **not** recommended.

#### **B.10.5 How do I edit an existing Relationship?**

**Open** the “**Relationships**” pane (click *B.10.1*).

**View** the Relationship’s **properties** (click *B.10.3*).

**Configure** the Relationship’s **properties** (click *D.9.2*).

**Close** the “**Relationships**” pane (click *B.10.8*).

### B.10.6 How do I create a new Relationship?

If you want to **create** a Relationship, you may click “*D.9.1 How do I create a new Relationship?*”.

### B.10.7 How do I delete a Relationship?

Open the “**Relationships**” pane (click *B.10.1*).

If it is not unhidden, you unhide the Relationship (click *B.10.4.2*) that you want to **delete**. You can then **delete** the Relationship in either of the following ways:

- Right-click on one of the **lines** that represents the Relationship and click on “**Delete**” from the pop-up menu.
- Select the Relationship by clicking on one of the **lines** that represents it (the line will become **highlighted in bold**), and then press the “**Supr**” key.


In both cases above, when I say you (double)click on a **line** you have to do it with the mouse placed **on the straight body of the line, and not on either of the crooked endpoints**. If you **click on either crooked endpoint of a line, this will be as clicking on the background** of the “**Relationships**” pane. Notice also the lines are narrow, and if you do not place the mouse correctly over the **line, you will be clicking on the background** of the “**Relationships**” pane, getting a different outcome.

When you **delete** a Relationship, a pop-up confirmation box will be shown: if you click on “**Yes**”, the Relationship is then deleted, and cannot be recovered (there is no paper trash or undo for this).

When you are done deleting Relationships, **close** the “**Relationships**” pane (click *B.10.8*).

### B.10.8 How do I close the “**Relationships**” pane?

You close the “**Relationships**” pane in either of the following ways:

- Click on the **Close “X”** icon in the **tab** of the “**Relationships**” pane.
- Click on the **Close**  icon placed on right side of the “**Design**” contextual Ribbon. If the “**Design**” Ribbon is not currently shown, click on “**Design**” from the “**Ribbon-bar**” (below the “**Relationship Tools**” contextual label).
- Right-click on the **tab** of the “**Relationships**” pane and click on either “**C**lose” or “**C**lose All” from the pop-up menu.
- Right-click anywhere on the gray background of the “**Relationships**” pane and click on “**C**lose” from the pop-up menu.

Regardless of the way you close the “**Relationship Tools**” pane, if it contains **unsaved layout changes**, MS-Access will show a dialogue-box asking if you want to save the changes to the layout. This dialog box has the buttons “**Yes**”, “**No**” and “**Cancel**”:







- “**Yes**” **saves** the **layout** and **closes** the “**Relationships**” pane.
- “**No**” **does not save** the **layout** and **closes** the “**Relationships**” pane.
- “**Cancel**” **does not save** the **layout** and  **Cancels** the closing of the “**Relationships**”

pane.

### **B.11 Can I use a drop-down/expression menu even if its icon is not shown?**

Yes, if you **click** on the **place** where the drop-down/expression **menu icon** is displayed, the **menu** will be **shown**, even if the **menu icon** is **not shown**. For the case of property cells, if you click on the place where the menu icon is displayed, the **menu** will be **shown** even **without** having previously **selected** the cell.

This capacity just described applies to:

- **Row property drop-down menu** “” icon
- **Row expression menu** “” icon in rows
- **Cell value drop-down menu** “” icon
- **Cell property drop-down menu** “” icon
- **Cell configuration drop-down menu** “” icon
- **Cell expression menu** “” icon

Clicking **directly** on the **rightmost side** of the cell/row to show the menu **saves you one click** every time you want to modify a value/property/value/expression. This is quite useful both for entering data and for modifying your database design.



## PART C. CONCEPTS AND INTERNALS OF DATABASES

You may click:

- “C.1 What are the main concepts of databases?”
- “C.2 What are objects, names, keywords, data types, constants, variables, operators, functions and expressions?”
- “C.3 What are fields, field value-lists, records and record-lists?”
- “C.4 What are database Tables?”
- “C.5 What is a pointer?”
- “C.6 What is a Null?”
- “C.7 What are duplicate records and duplicate field values?”
- “C.8 What is indexing?”
- “C.9 How do I prevent duplicate field values and duplicate records?”
- “C.10 What are the Table Key(s) and how should I handle them?”
- “C.11 What is a Relationship?”

### C.1 What are the main concepts of databases?

You may click:

- “C.1.1 What is a database, a record and a field?”
- “C.1.2 What is the difference between a database and a spreadsheet?”
- “C.1.3 What is a database design?”
- “C.1.4 What is an SQL Query?”
- “C.1.5 What is the Structured Query Language (SQL)?”
- “C.1.6 How is a database used?”

#### C.1.1 What is a database, a record and a field?

A database is a large collection of **data** stored in a **structured** and **organized** way, plus, a collection of **functions** (called **Queries**) to produce useful **reports** and **summaries** from the database data. In each Query you may define, in a reasonable simple way, very complex operations over the database in order to produce very different types of reports, listings, summaries, statistics and any processed data you may need out of the database. More advanced Queries also allow you to automatically modify/update the data stored in the database.

Conceptually, data in a database is stored and processed as **record-lists**.

A **record-list** is a **list** of **records**, such that each and every record in the **record-list** has the same **ordered list of fields**. Each record-list stores a **variable** number of records, and records are stored in **no particular order** (although they can be ordered when listing them).

A **record** is an **ordered list of fields**, each field in the record having a **different field name**, a specific **data type** and some other field **properties**. All the records in each record-list have always **the same** ordered list of **fields**, fields **names** and field **types**.

You can think of a record-list as a **table** with a **fixed** number of columns (the fields), and a **variable** number of rows (the records). If you have a record-list of invoices, you visualize a table in which each **row** in the table is **one record** with the data corresponding to one concrete invoice and **each column** is a **list of values** for one specific **field name** of the invoices (invoice date, invoice amount, invoice VAT, ...). You can have **many** different tables in the database, each with **its own specific structure**.

If you currently handle different types of data (list of employees, list of customers, list of invoices, list of bank account balances, list of suppliers, list of orders, ... ) in a sparse way (e.g., in different excel files, word documents, ...), you may **consolidate** everything **in a single database**, with a set of suitable database Tables. The advantage is not only having all the information in a single file, and in a homogeneous format, but much more interesting, that you can **process and exploit** all that information in your Tables obtaining valuable summaries, reports, statistics, forecasts, etc. by writing your own Queries.

### **C.1.2 What is the difference between a database and a spreadsheet?**

The main overall differences between a spreadsheet and a database are:

- Databases can **relate** the values of one Table column to the values **of another column on another Table**, allowing to cross reference the values between all the Tables in a very **sound and error-free way**. Spreadsheets **cannot** do this.
- Databases impose that **all the values in each column** of each of its Tables (sheets) have the **same** data type and interpretation (semantics), while in a spreadsheet you can do whatever you want.
- Databases can **check** for you that the values in **certain columns** are **unique**, while a spreadsheet cannot do it. This is useful!
- Databases separate the **code** (Queries) to process the data from the data itself (Tables) making it much more difficult to unknowingly modify the code. Spreadsheets **mix** cells with **data** and **formulas throughout** the sheet: **very flexible**, but **very risky**.
- Databases require **considerably more initial work** than a spreadsheet before you can start using them and are **more rigid** in the way you use them. However, databases are **much safer** than spreadsheets in terms of accidental errors, and can handle much **larger amounts of data**, and **more diverse**, than using a collection of spreadsheets.

In a spreadsheet (from Excel, Google sheet, or a similar application), you refer to values by the **coordinates** in the spreadsheet. You usually **mix** cells with **stored values** with cells with **formulas** the way you want and **without any restriction**. Spreadsheets are incredibly **flexible** and **easy to write**, and also, allow you to do a quick solution for what you want to calculate or process. However, a spreadsheet is not a good tool to store and process large amounts of data in a professional way because **mixing up the data and the code** (i.e., the formulas) is **dangerous**, and its **flexibility** implies that it is quite **easy**

**to introduce errors in the data or calculations.** Also, it is **not easy** to properly **connect and combine different spreadsheets**, so if you want a **global coherent view** of large and diverse types of data, **spreadsheets do not work well.**

You can think of a database somehow as large number of spreadsheets, each of them with homogeneous column values, that can be connected and that work in a totally coherent and homogeneous way. In a database, you would have some **spreadsheets (with homogeneous column values) to store** the information, following fixed patterns (these are **called Tables**), and you would have other **“spreadsheets” to do whatever processing** you want over the stored data (these are **called Queries**). This is, in a database **the stored data and the code** the manipulates the data is **totally separated.** Also, the **code** that processes the data is **hidden** inside the Query objects, to avoid accidental manipulation of the code. Unlike spreadsheets that are totally flexible in the way you want to structure and store your information, databases require that the stored data follows some specific structure and rules (described in this **Lightning Guide**). The most fundamental restriction is that in database **Tables all the cells** (fields) in each **column** must have the **same data type and value interpretation** (semantics).

If you **really like** spreadsheets, you will **just love** databases!!

### **C.1.3 What is a database design?**

The database **design** consists of:

- Defining the database **Tables** (click *C.4*)
- Defining the **Relationships** between Tables (click *C.11*).

If you want more detail on database design, you may click **“Part D. Designing my databases with MS-Access”**.

### **C.1.4 What is an SQL Query?**

If at this moment you are reading this **Lightning Guide linearly**, click to read **“F.4.2 What is an SQL Query?”** and then return here (you return by simultaneously pressing the **“Alt”** and **“←”** keys).

### **C.1.5 What is the Structured Query Language (SQL)**

If at this moment you are reading this **Lightning Guide linearly**, click to read **“F.1 What is the Structured Query Language (SQL)?”** and then return here (you return by simultaneously pressing the **“Alt”** and **“←”** keys).

### **C.1.6 How is a database used?**

Using a database consists of **inputting data** records (click *Part E*) into the database Tables, **coding Queries** (click *Part F*) to exploit the data in the database, and **running the Queries** (click *B.4.1.9*) to obtain the desired summaries, statistics, reports, etc. from your database.

When you first create the database, you usually upload to it some amount of current and past information records. This usually implies reformatting existing information to adapt it to the design of the database and pasting it (click *E.5*) in blocks to the different database Tables. When doing this you have to be careful with the consistency of the past information records (click *E.9*) that you are uploading.

You then code some initial Queries to obtain relevant reports out of the database. One Query may provide total expenses by category per year, another one the listing of employees in any given year, or whatever complex information you may want to get.

The lifecycle of the database (click *Part I*) continues as you input records arising from the day to day activity, and you further create/modify/delete Queries as you see fit to suit the information you want to get from the database.

## **C.2 What are objects, names, keywords, data types, constants, variables, operators, functions and expressions?**

You may click:

- “C.2.1 What is an object and a name?”
- “C.2.3 What is a keyword?”
- “C.2.4 What is a data type?”
- “C.2.5 What is a constant?”
- “C.2.6 What is a variable?”
- “C.2.7 What is an operator?”
- “C.2.8 What is a function?”
- “C.2.9 What is an expression?”
- “C.2.10 What is an SQL operation?”
- “C.2.11 What is expression syntax?”

### **C.2.1 What is an object and a name?**

A **name** (also called “**alias**”) is a **text string** that serves to **reference** your database **objects** as well as the **properties** of objects. Some examples of database **objects** are **Tables, Queries** and **SQL operations** within your Query code. Some examples of object **properties** are the **fields**, that are a property of Tables, Queries and SQL operations.

If you want to know more about the importance of assigning **proper** names to your Tables, Queries, fields and other database objects/properties, you may click “*D.2 How do I carefully assign good names from the very beginning?*”.

### **C.2.2 What is a qualified field name?**

In some database scopes (e.g., within your SQL code) a field name may be ambiguous, because you have **two** fields with the **same name** and in the same scope. In this case, to distinguish both fields, each equal field name is **prefixed** with the object name to which each of the two fields belongs. Each field name and its prefix object name are separated with a period “.” character. The **prefixing** object **name** is called the “**qualifier**” and the **resulting** field name is called a “**qualified field name**”.

For example, if you have two Tables “**US\_Data**” and “**EU\_Data**” that each contain the same field name “**Yearly\_Sales**”, and you want to use **both** fields in the same SQL scope (e.g., in the same **Join** operation), then you have to prefix the field name by the Table

name, getting “**US\_Data.Yearly\_Sales**” and “**EU\_Data.Yearly\_Sales**”. These two **qualified** field names are not ambiguous, and it is now perfectly clear which specific field you are referring to in your SQL code.

Some other examples of qualified field names are:

```
Car_parts.Windshield_model  
Customer_Order.Windshield_model  
Capital_Cities.City  
City_Rainfal.City
```

If you want to know more about qualifying names, you may click “[D.2.5 What are the MS-Access formal rules for identifiers?](#)”.

### **C.2.3 What is a keyword?**

A **keyword** is a **name** that MS-Access has assigned to his **system objects, operators and built-in functions**. The SQL operators (e.g., “**SELECT**”, “**FROM**”, “**WHERE**”, “**IN**”, “**AND**”, ...) are keywords. Built-in function names (e.g., `Iif()`, `Log()`, `Switch()`, ...) are keywords. VBA data type identifiers (e.g., **Double**, **Date**, **Integer**, ...) are also keywords. MS-Access ignores case in keywords, as it does in names and text strings (click [A.4.3](#)). For example, “**SELECT**”, “**select**”, “**Select**” and “**SeLeCt**” are considered exactly the same keyword by MS-Access.

MS-Access **does not allow** you to use keywords as names **whenever** this could create an identifier conflict. You can use a **longer name** that **includes a keyword** inside it without any problem, but you should avoid names that are too similar to a keyword to avoid poorly readable SQL code.

### **C.2.4 What is a data type?**

A data type is the way to interpret generic binary information stored in the computer as a **specific type of information**. The different **specific types of information** are the different data types. In MS-Access you have data types for integer values, for fractional values, for date and time, for **Boolean** values, for text strings and to represent other types of information.

In MS-Access you have **two different sets** of data types. One is the **Table field types** and the other one is the **VBA data types**. The **Table field types** are the data types of the **fields** storing the information in your database **Tables**. The **VBA data types** are the data types used in your Query fields, in the expressions in your SQL code (i.e., your Queries) and in the expressions in your VBA code (i.e., your user-defined functions).

Throughout this Lightning Guide, Table field types are denoted in bold Calibri font (e.g., **Number-Integer**, **Number-Double**, **Short Text**, **Yes/No**, **Date/Time**, etc.), while VBA data types are denoted in bold italic Times font (e.g., *Integer*, *Double*, *String*, *Boolean*, *Date*, etc.). The **Table field types** and the **VBA data types** have **slightly different** names, but there is an **exact one-to-one equivalence** between **most** (not all) of them. If you want to know more about this equivalence, you may click “[G.2 How do I manage VBA data types and Table field types-sizes?](#)”.

### **C.2.5 What is a constant?**

A constant is a concrete **fixed value**. You can have constants of different Table field types and VBA data types. I briefly summarize here some MS-Access rules to represent

constants:

- **Boolean constants:**  
Values *True* and *False* (as such, **not** enclosed in quotes, as it is done for text string constants).
- **Integer number constants:**  
The usual way to write positive and negative integer numbers in decimal notation. Some examples of integer constants are: **123**, **456**, **0**, and **-15**.
- **Fractional number constants:**  
The usual way to write positive and negative fractional numbers in decimal notation. You can also use scientific notation. Some examples of fractional constants are: **4.0346**, **730**, **-3.234** and **3.5656E-23**.
- **Date and time constants:**  
Notice that both the **Date/Time** Table field type and the *Date* VBA data type **always** store a **date-part** and a **time-part**. However, you may have constants where the date-part and/or the time-part is **zero**, as follows:
  - **Zero-time** constants are **Date/Time** or *Date* constants where the **time-part** of the value is **zero**. One way of typing them is month number, day number and year number, separated by “/” and enclosed between “#”. Some examples are: **#3/30/2017#** and **#12/31/87#**
  - **Zero-date** constants are “**Date/Time** or *Date* constants where the **date-part** of the value is **zero**. One way of writing them is: hour (0 to 23) and minutes (and optionally seconds) separated by “:”. Some examples are: **#23:59#**, **#23:59:46#**
  - **Date-and-time** constants are **Date/Time** or *Date* constants that have a non-zero date-part and a non-zero time-part. One way of writing them is writing the date-part and the time-part separated by a blank “ ”, and all enclosed between “#”. Some examples are: **#3/30/2017 23:59#**, **#1/28/2017 23:59:46#**

Notice that MS-Access will check that date constants correspond to a correct calendar/clock value. If you use a constant of a non-existing date (e.g., **#2/29/1987#** or **#11/31/2019#**) or time (e.g., **#24:00#**, **#23:60#**), this will not be accepted by MS-Access, and will report an error.
- **Text string constants:**  
A string of characters enclosed in **double quotes** or in **single quotes**. Some examples are: **"Overlap dates"**, **'US-Cities'**, **"Engine\_Parts"**, and **'Delivery\_Date'**.  
Notice that in VBA code you **cannot use single quotes** to enclose text strings.

If you want to know more about MS-Access constants, you may click “[G.4 How do I write a constant?](#)”.

### C.2.6 What is a variable?

A variable is a **name** that represents a **generic value**. Variables usually belong to **one specific** data type, although in VBA they can also be *Variant*, which is a kind of wildcard “generic” data type.

### C.2.7 What is an operator?

An operator is a **named** mathematical computation over some input variables that

returns one result. Each time you apply the operator to some concrete input values it produces one single concrete output value. Most operators work over either **one** input variable (also called unary operator) or **two** input variables (also called binary operator). The input variables of an operator are usually called “operands”. Some examples of operators are: “**AND**”, “**OR**”, “**+**”, “**-**”, “**/**” and “**\***”.

Operators are **deterministic** and do not have memory. This means that if you use the operator with the same input values, you **always** get **the same** result.

The way to use an operator is by writing its name usually **before** its input variable (if it only takes one input value) or writing its name **between** its two input variables (if it takes two input values). Some examples of operators with two input values are the following:

```
3 + 4
34.5 * 36.34
True AND True
```

Each value operator has a specific data type (or **compatible** ones) for each of its operands and for its output result. For example, the Comparison operator “**<**” may take two *String* operands and produces a *Boolean* result. Some data types are **compatible**, allowing them to be combined. For example, with the “**+**” operator you can add an **integer-like** value with a **fractional** value, and you can also add a **fractional** value with a **Date/Time** value.

If you want to know more about MS-Access value operators, you may click “[G.1 What are the main differences between the three expression scopes?](#)”

### C.2.8 What is a function?

A function is a **named** mathematical computation that takes some **variables** as input and produces **one single output value** as a result. Each time you apply the function with some concrete input values, the function is computed, and it returns **one** single concrete **output** value. The input variables of a function are usually called “arguments”.

Functions (with **very few** exceptions) are **deterministic** and do not have memory. This means that if you use the function over the same input values, you **always** get **the same** result.

The way to use (also called “invoke”, “instantiate” or “run”) a function is by writing its name followed by its input values separated by commas and enclosing all of the input values in a pair of parentheses. Some examples are:

```
Round(10.45, 2)
Is_Null("Ford_Mustang")
Overlap_dates(#3/1/1980#, #6/23/1982#, #1/1/1981#, #12/31/1981#)
```

Most functions belong to **one** data type, corresponding to the data type of **the value that it returns**. For example, *Boolean* functions are the ones that produce a *Boolean* value, and *Integer* functions are the ones that produce an *Integer* value. Some examples of functions, and their data type are:

```
Round(Floating_variable, decimal_positions)      (returns an Integer)
Is_Null(Input_Variable)                          (returns a Boolean)
Iif(Age>0, (Height/Age)^2, "Error")              (returns a Variant)
```

Each input variable of a function may belong to a different data type. You may check



the three example functions above and see that they contain input values of different data types. Most functions have a **fixed** number of input variables. Most functions have input variables **each** of them belonging to a **specific** data type.

Notice however that there are some functions that return a **Variant** (see description above on VBA data types) and/or that have one or more input variables of variant type. It is also possible to have functions with a **variable** number of input variables.

If you want to know more about MS-Access functions, you may click “*G.6 How do I use functions in an expression?*”.

### C.2.9 What is an expression?

An **expression** is a mathematical **formula** composed of **variables** combined with **constants**, **operators** and/or **functions** that has **parts** of it enclosed between **parentheses**. When you replace the **variables** by **concrete input values**, the **expression** produces **one single result**. Each time you compute the **expression** over some **input values** for its **variables**, the expression returns **one single concrete** result.

**Expressions** (unless they contain non-deterministic functions) are **deterministic** and do not have memory. This means that if you compute an expression over **the same** input values, you **always** get **the same** result.

An **expression** belongs to one **data type**, corresponding to the **data type** of the **value** that it **returns**. For example, **Boolean** expressions are the ones that produce a **Boolean True** or **False** value, and integer expression are the ones that produce an **Integer** value. An **expression** can also return a variant data type. Some examples of expressions, and their data type are:

```
Iif(My_Date >= Date()-180,From-To,Date()+34)    (returns a Date)
(Age > 0) and (Age < 120)                      (returns a Boolean)
Iif(Age>0,Average((Height/Age)^2,-1)          (returns a Double)
```

The constants, variables, operators and functions in an expression can be of **different** data types. You may check the three example expressions above and see that they contain constants, variables, operators and functions of different data types.

An **expression** may return **different results** depending on the **computation order** (also called evaluation order). For example, the expression “(3\*4)+2” produces **14** but it produces **18** if I change the parentheses to “3\*(4+2)”. The evaluation order is determined by the parentheses. When parentheses are not used, expressions have specific **default** evaluation rules. Expressions are normally evaluated **left to right**, but some operators have higher evaluation **precedence**. For example, “\*” has precedence over “+”. My advice is that, unless you are **very sure** of the evaluation rules for an expression, you **should write parentheses** to indicate explicitly the **evaluation order** that you want for it.

If you want to know more about MS-Access SQL operations, you may check different chapters/sections clicking on “*Part F. Writing SQL Queries to use my database*”. If you want to know more about MS-Access value expressions, you may check different chapters/sections clicking on “*Part G. Writing expressions*”.

### C.2.10 What is an SQL operation?

An “**SQL operation**” is a mathematical **formula** composed of record-list **variables** (Query name or Table names) combined with record-list **operators** (SQL operators) and

**functions** that has **parts** of it enclosed between **parentheses**. When you replace the record-list **variables** by **concrete input values**, the **SQL operation** produces **one single resulting record-list**.

You probably have noticed that this is **the same** definition as the one for an expression, that I used in the previous section. I am using the term SQL “**operation**” instead of SQL “**expression**” to avoid having to constantly clarify along this Guide if an “expression” is a “SQL expression” or a “value expression”. Moreover, the term “SQL expression” is ambiguous, because it can refer **both** to an expression built using the SQL operators (e.g., a **Select** operation) and to a value expression used inside your SQL code (e.g., “**Log (x)**”).

Conceptually, both SQL operations and expressions are just **expressions**. However, the way to design and write an SQL operation and an expression is **very** different. SQL operations are written using the **SQL language**, while expression are written using more conventional operators and functions, like “+”, “\*”, “**Max ()**” or “**Round()**”.

If you want to know more about MS-Access SQL operations, you may check different chapter/sections clicking on “*Part F. Writing SQL Queries to use my database*”. If you want to know more about MS-Access value expressions, you may check different chapter/sections clicking on “*Part G. Writing expressions*”.

### **C.2.11 What is expression syntax?**

The expression **syntax** are the **rules** that indicate how to **write correctly** an expression. Each expression is written according to some rules called **expression syntax**. These rules include what operators you may use (like “+”, “-”, “/”, “**AND**”, “**OR**”, ...), what functions you may use (like “**Average()**”, “**Date()**”, “**Year()**”, “**If()**”, ...), how to make references to variables, how to write constants, and other rules (parentheses, identifiers, ...) that state how an expression can be written. If you write an expression that does not conform to the expression syntax, MS-Access will show an error message with some indication of the cause of the error. A simple example of a syntax error is wrong parentheses matching (i.e., the expression has a different number of open parentheses and close parentheses).

MS-Access has **different** expression **syntax** in its **SQL operations**, in its **VBA expressions** and in its **Table “Design View” expressions**. You should be very aware of this fact. If you want to know more about this, you may click “*G.1 What are the main differences between the three expression scopes?*”.

If you want to know more about MS-Access SQL operation syntax, you may check different chapters/sections clicking on “*Part F. Writing SQL Queries to use my database*”. If you want to know more about MS-Access expression syntax, you may check different chapters/sections clicking on “*Part G. Writing expressions*”.

## **C.3 What are fields, field value-lists, records and record-lists?**

You may click:

- “*C.3.1 What are fields, field names and field types?*”
- “*C.3.2 What is a record and a record type?*”
- “*C.3.3 What is a record-list?*”

- “C.3.4 What is a field value-list?”

### C.3.1 What are fields, field names and field types?

In a database the smallest element of information is the **field**. Each field stores **one and only one value**<sup>19</sup>, has a **field name** (click C.2.1) to identify it, and has an associated **data type** which is called **field type** (e.g., **Short Text, Number, Date/Time, ...**).

The value stored by the field **always** corresponds to its **field type**. The field may also be **empty**, in which case it is said to store the special element “**Null**” (click C.6). **Field types** and **Nulls** are **both** very important for the correct processing in the database. If you want to know more about configuring field types, you may click “D.4 How do I configure a Table field data type and size?”.

A few examples of **field names, field types** and **field values** are:

Field Name =>	Invoice_Date	Field Name =>	Invoice_Num	Field Name =>	Amount
Field Type =>	Date/Time	Field Type =>	Number-Long	Field Type =>	Number-Double
Field Value =>	03-jan-2015	Field Value =>	125	Field Value =>	457.56

### C.3.2 What is a record and a record type?

A **record** is a **fixed ordered list** of **fields**, each field with a **different field name**. An example of a record is:

Invoice_Date	Invoice_Num	Amount	Invoice_VAT	Description	Customer
Date/Time	Number-Long	Number-Double	Number-Double	Short Text	Short Text
03-ene-15	501	457,56	96,09	Transceiver	Cisco

The **field names** in any given record **must be different** to avoid ambiguity when referring to them in the SQL code of Queries. However, the **field types** in a record do not need to be unique and may be repeated one or more times, as needed. The field names in **different** records may be repeated without causing any problem, and actually this is usual practice.

The **record type** is the **ordered list** of **field types** in the record. The record type of the record example above is:

Date/Time	Number-Long	Number-Double	Number-Double	Short Text	Short Text
-----------	-------------	---------------	---------------	------------	------------

You may see in this example that a record can have duplicated **field types**, as I already mentioned.

### C.3.3 What is a record-list?

A **record-list** is a list of records, all of which have **the same** fixed ordered list of **field names**, and the same **record type**. A **record-list** is therefore a list of homogeneous records. Record-lists are quite dynamic, and it is frequent to add and remove records

<sup>19</sup> There is an exception to this, allowing for storage of **lists** of **Number** and **Short Text** values, but its usage is very unfrequent.

from any record-list as a consequence of database processing.

Let me show you an example of a record-list:

Invoice_Date	Invoice_Num	Amount	Invoice_VAT	Description	Customer
Date/Time	Number-Long	Number-Double	Number-Double	Short Text	Short Text
03-ene-15	501	457,56	96,09	Transceiver	Cisco
04-sep-15	503	1.435,34	301,42	Wifi Access Point	NEC
08-mar-15	502	747,67	157,01	Wifi Access Point	Telefonica
08-jul-15	504	1.335,00	280,35	120 cans	Tata
07-ene-16	505	1.892,90	397,51	IP Switch	Huawei
26-oct-16	506	4.125,89	866,44	2000 SIM cards	Telefonica
26-feb-16	507	144,70	30,39	130 cans	Tata
13-ene-17	508	2.690,55	565,02	131 cans	Huawei
18-dic-17	509	5.160,00	1.083,60	IP Switch	Tata
15-feb-17	510	2.243,01	471,03	Consulting Services	Cisco
20-ene-18	512	1.058,50	222,29	IP Switch	NEC
05-oct-18	511	1.695,56	356,07	1678 SIM cards	Telefonica
05-feb-18	513	6.540,04	1.373,41	Wifi Access Point	NEC

All the information in the database is stored and processed as **record-lists**. Each database Table stores **one** record-list. Each database Query, written in SQL, takes **one or more record-lists as input**, and produces as a result **one** new specific **output record-list**. A Query is therefore like a function that takes **one or more** record-lists as arguments and produces **one and only** one output record-list as a result. The SQL language allows a very powerful and flexible way to select, merge, combine, split, and perform many other operations over record-lists. In summary, the concept of **record-list** is **absolutely fundamental** in databases!

You may visualize a record-list as a table, a record as a row in the table, a field-list as a column in the table, and a field value as a cell in the table. However, calling “table” to record-lists creates great ambiguity between a “table” (a record-list) and a “Table” of the database, which are similar, but not the same thing. A database **Table** is a very specific database object, and it **stores** a record-list, but it is **not the same** as a record-list (a database Table has very specific properties and characteristics). At the same time, **not** every record-list is a database Table. For this reason, throughout this **Lightning Guide** I will specifically use the term **record-list** to refer to a record-list and the term **Table** (with capital “T”) to refer to a database Table.

### C.3.4 What is a field value-list?

In most books the term **field** is used to denote either of:

- The field **name**
- **One value** element in **one specific record**
- **The list of all field values** belonging to a **given field** within a **record-list**.

To avoid this undesirable ambiguity, I will mostly use the term “**field name**” to denote the field name, the term “**field value**” to denote **one value of one record**, and the specific term “**field value-list**” to denote the **list of all field values** belonging to a **given field** within a **given record-list**.

Notice that a given **field name** of a record-list applies **both** to the corresponding **field-list** of the record-list, and to **each and every field value** belonging to that field value-list.

## **C.4 What are database Tables?**

Each record-list **permanently stored** (i.e., not just temporally used along Query processing) in a database is stored in a **Table**. In addition to storing a record-list, a Table has some important configured properties over its fields. When designing a database, **defining the Tables** that will store the record-lists of the database is a **fundamental design decision**. When you define the set of Tables of your database you should carefully think what are the “entities” you want to represent in each Table, what are the relevant data elements that make a proper description of each entity, and what are the corresponding field **properties** in the Table. In a more operative way, when defining each Table you need to define the names of its fields, the data type of each field, and the order of the fields. In addition to this very basic definition of the Table, you will usually also define the **Key** fields of the table, **indexed** fields, value masks, Relationships with other database Tables and a few other advanced properties.

If you define a database, and at a later stage you want to change the Tables (add fields, remove fields, change properties modify Relationships, ...) this will be possible, but, it will require to adjust a number of Queries and other database characteristics because they will be affected by the changes in the Table design. This is why you should devote **some time and thought** to define the database Tables in the initial database design, and also, devote **some time and thought every time** you want to modify them.

If you want to configure your Tables in MS-Access, you may click “*D.3 How do I create and design a Table and its fields?*”.

## **C.5 What is a pointer?**

A **pointer** is an **internal storage address** of the database that allows the system to locate each **record** and **field value** stored in the database.

The system uses **record pointers** to “move”, “order” and “copy” the **records**, without actually moving, ordering or copying them. What the system does is move, order or copy **the record pointers**, but **not** the record data itself. The system also uses the record pointer whenever it needs to access the actual record values, for example, to show them to you in the screen, or to do a calculation over them.

The system also uses **field pointers**, in addition to **record pointers**. In this way, the content of a record **is not** the actual list of values, but a **list of field pointers** each pointing to the corresponding **field value**.

You can think of a pointer as a kind of **web-link** (i.e., a web URL). You can **copy the web-link, mail it** to a friend, insert it **in a document**, include it in your list of favorites, and in the history of the navigator, **but, the actual webpage stays put in its web server**. You only copy, move, duplicate or store the **link** (“**pointer**”) to the webpage, and **not** the webpage itself.

Handling data with pointers has lots of advantages. When you have the same value (e.g., an e-mail address, a person name, ...) in different records of the database, that value (on most cases) **is not stored twice** in the database storage, the system just duplicates the

**field pointer**, saving storage and avoiding possible data incoherence. Also, when the system processes the records (e.g., change the order of the record-list according to the value of a specific field) or runs a Query to get some results, the system will **not actually copy the records nor move them around** in the system storage, and rather will just **copy or move** the corresponding record pointers.

You will **not** handle pointers when you work with a database, because it is an internal system tool. However, understanding what a pointer is **will be very useful** for you to understand **indexing** and **Relationships**, which are very relevant when working with a database.

Chapter “C.8 What is indexing?” presents some of the advantages of handling data based on pointers.

## C.6 What is a Null?

You may think of a **Null** as an **unknown** value.

Actually, a **Null** is a **field pointer** pointing to an invalid place (e.g., non-existing place) in the system storage. Recall that a **field pointer** is an **internal** storage address of the database that allows the system to locate **field values** stored in the database. Therefore, a **Null** is a pointer to a **non-existing** storage place. Since the database system processes fields by handling their corresponding field pointers, whenever the field pointer is not valid, then the system is handling a **Null**.

A **Null is not a valid value** (and it is **not the neutral element**). A **Number** field containing the value “0” **is not a Null**. The **zero-length** string (i.e., the string containing zero characters) **is not a Null**. A string **only** composed of **invisible characters** (e.g., space, tab, new-line, ...) **is not a Null**. The case of *String* fields is particularly problematic, because a **Null**, a zero-length string and an invisible string **are all different**, but they all **look the same as an empty cell** on the screen.

Because a **Null** is like an **unknown** value, this may cause very different effects in your database operators and functions. Fields having **Null** are treated as **containing the same value** for the purpose of **duplicate records** because they are considered “**equally unknown**”. However, comparing two **Nulls** with the “=” operator returns **Null** (it **does not** return *True*), because being both **unknown** the result of asking if they are the same is **also** unknown. **Almost all** value operators (click G.5) with one (or both) **Null** operand(s) return **Null**. However, there are **a few exceptions** in which the value operator returns a **valid value**. **Most** built-in functions with one (or more) **Null** arguments return **Null**. However, there are **some** built-in functions with one (or more) **Null** arguments that will return a **valid value**, while **others** will **crash**.

In spite of **Nulls** being **highly undesirable**, **Nulls** are **unavoidable**. Even if none of your Tables have **Nulls**, **outer join SQL operators, functions** and other circumstances will most likely **generate Nulls**, whether you like it or not. Therefore, you need to know what a **Null** is, what it implies, and how to properly handle it.

If you want to know more about the different effects of **Nulls** and how to handle them, you may click “K.5 Why and how should I carefully handle Nulls in my Queries?”

## C.7 What are duplicate records and duplicate field values?

You may click:

- “C.7.1 What are duplicate records?”
- “C.7.2 What are duplicate field values?”

### C.7.1 What are duplicate records?

**Duplicate records** are records from the **same record-list** that have the **same** value (or **Null**) in **all their field names**, field name to field name. In other words, if we compare their values **field name by field**, name they have the same value (or **Null**) in all of the field names. Notice that **for the purpose of duplicate** records two **Null** are considered as the same value.

Let us check duplicate records in the following example record-list:

T_Temperatures_A					
Capital	District	Cal_Year	Quart	Temp_max	Temp_min
Brasilia	Asa_Norte	2018	Q1	17.5	
Brasilia	Asa_Norte	2018	Q1	17.5	
Brasilia	Asa_Norte	2018	Q1		17.5
Sao Paulo	Jardins	2018	Q1	17.5	
Brasilia	Asa_Sul	2018	Q1	17.5	
Brasilia	Asa_Norte	2019	Q1	17.5	
Brasilia	Asa_Norte	2018	Q2	17.5	
Brasilia	Asa_Norte	2018	Q1	22.7	
Brasilia	Asa_Norte	2018	Q1	17.5	11.5

The first two records (shaded in light green) are **duplicate records**, because they have the same values (or **Null**) under the **same field names**. The third record is **not** a duplicate record because it has **exactly the same values** (or **Null**) as the first two records, but, it is **not under the same field names**. Remaining records are **not** duplicate records because each of them has **one** field with a different value from the first two records. I have shaded in light orange the field values that are different in the records that are not duplicate records.

Obviously, you can have **more than two duplicate records** in a given record-list.

Duplicate records are undesirable on most occasions, because sometimes are openly wrong, and sometimes create ambiguity between an invalid and a valid record. One example of being openly wrong would be the case of a Table of capital cities (with just the capital name as its single field). There are no two capital cities with the same name, so if you have two records with the same name, it is an error. One example of creating ambiguity is a Table of invoices with fields “Invoice\_Date”, “Amount” and “Customer”. You could have several invoices in the same day for the same customer, and if you see two with the same amount, you would be in doubt if it is really two invoices, or it was duplicated by mistake. It would be better practice to add a field like “Invoice\_number” that makes **every record** in the invoice Table **unique**.



Having, or not having, duplicate records in a record-list is a very important question that will arise when analyzing other aspects of databases. For each record-list that you will be handling in the database, either as a Table or a Query result, it is very important that you are **perfectly aware** about whether it **may** have, or **may not** have, duplicate records.

In the **large majority** (but not all) of **Tables** you **will not** want duplicate records. You can do a good database design and configure Table properties and the system will **automatically** prevent duplicates. If you want to know more about how to prevent duplicate records in Tables with MS-Access, you may click “[C.9 How do I prevent duplicate field values and duplicate records?](#)” and “[D.6 How do I configure the Primary Key field\(s\) of a Table?](#)”.

Regarding record-lists produced by **Queries**, in particular during intermediate processing, **they may have** duplicate records. Even if **all** the database Tables are duplicate-free, the Query processing **may generate** duplicate records. The SQL language has ways to remove duplicates (the “**DISTINCT**” clause and the “**UNION**” operator) in case you do not want duplicates in a given output record-list.

### C.7.2 What are duplicate field values?

Duplicate field values are **records** from the **same record-list** that have the **same values** in **a group of field names**, field name to field name. In other words, if we compare their values field name by field name (for field names belonging to the group) **between the two records**, they all have the same value. Notice that in this definition **Null is not considered the same value**, while in the definition for duplicate records **Null is considered as the same value**.

Let us check duplicate field values over the group of field names “Capital”, “District” and “Cal\_Year” in the following example record-list:

T_Temperatures_B					
Capital	District	Cal_Year	Quart	Temp_max	Temp_min
Brasilia	Asa_Norte	2018	Q1	17.5	
Brasilia	Asa_Norte	2018	Q2	17.5	
Sao Paulo	Centro	2018	Q2	17.0	
Brasilia	Asa_Sul	2018	Q3	17.5	
Brasilia	Asa_Norte	2018	Q4	22.7	
Brasilia	Asa_Norte		Q4	27.5	
Brasilia	Asa_Norte		Q1	18.5	

The first two records (shaded in green) have **duplicate field values over the group of field names** “Capital”, “District” and “Cal\_Year”, because they have the same values under the **same field names**. The third to fifth record are **not** duplicate field values of the first two records because they have a different value in either of the three field names. The third to fifth records are also not duplicate field values among themselves. The field with a different value in records third to fifth has been shaded in light orange.

The last two records are **not** duplicate values over “Capital”, “District” and “Cal\_Year”, because they have the same values under **field names** “Capital” and “District”, but, they **do not have the same value** under field name “Cal\_Year”. They both have **Null** under

“Cal\_Year”, and **Null is not considered the same value** for the purpose of duplicate field values. Recall that **Null is considered** the same value for the purpose of **duplicate records**.

Obviously, you can have **more than two** duplicate field values over a certain group of field names in a given record-list.

Notice that being duplicate-free over **all** the field names is **not the same** as being duplicate-free, because duplicate-free requires that **Nulls also are not the same**, while **Nulls are not** considered to be duplicates over a group of field names.

## C.8 What is indexing?

Indexing is a computer science technique that allows to get the result of certain operations over lists **much faster**. In particular, it allows to order a list instantaneously and to get a partition of a list over a given value in a very fast way.

An **index** is a list of **record pointers, ordered** on the **values** of **one** (or more) field value-list.

If you want to know more, you may click:

- “C.8.1 Why is indexing useful?”
- “C.8.2 Can I create one index over a list of field names (called a composite index)?”
- “C.8.3 What different types of indexes are there?”
- “C.8.4 What indexes can I configure in a given Table?”
- “C.8.5 Why should I use indexing in my database?”

If you want to know how to configure indexes in MS-Access, you may click:

- “D.5.1.7 What is the “Required” Table field property?”
- “D.5.1.8 What is the “Indexed” Table field property?”
- “D.7 How do I add simple and/or composite index(es) to a Table?”

Also, if you configure a **simple** or **composite Key** in the Table, MS-Access will **automatically** configure the **Key** field(s) as a **simple** or **composite** (respectively) index **without duplicate values and without Nulls**. If you want to know more about how to configure your **Key** fields, you may click:

- “D.6 How do I configure the Primary Key field(s) of a Table?”.

### C.8.1 Why is indexing useful?

In databases, indexing consists of building an **ordered list of pointers** (click C.5) to the records of a record-list, where the order of the pointers is based on the **values** of the record fields. An **index** is therefore a list of **record pointers, ordered** on the **values** of one (or more) field value-lists. The cost of indexing is that the system has to devote some **previous** processor time to build the index and also use some **memory** to store the index(es). The index is valid while the record-list is the same. If you modify the record-list by removing or inserting a record, you have to update the indexes.

Let us see the advantages of indexing. Imagine you have a record-list of invoices with

1,000,000 records. Now you want to run a Query with the following SQL code (if you want to know more about the **Select** operation, you may click “[F.7 What is a Select operation and how do I write it?](#)”):

```
SELECT Invoice_num, Invoice_Date, Amount
FROM Invoices_list
WHERE Amount < 3,000
```

Without indexing, the system would have to retrieve the **million** records from the database Table, and painfully **go over all of them**, checking **one by one** if the value of its “Amount” field is lower than 3,000 or not. In case it is lower, the record is included in the output record-list of the Query, and in case it is not, the record is not included. It is pretty clear that checking the value of 1,000,000 records **one by one** will take some time before getting the output record-list.

An excellent alternative to this **one-by-one** approach is to use **indexing** and **binary search**. If the Invoice record-list was **ordered**, from min to max value of “Amount”, you **do not need** to check the million records. Rather, you use a search method called **binary search, which is much faster**. To apply binary search to an indexed record-list, you first fetch the record in the middle of the record-list and check if its “Amount” value is higher or lower than 3,000. In case it is higher, you fetch and check the record in the position three quarters in the record-list, and in case is lower, you fetch and check the record in the position one quarter. You do the same a few times **until** you find **two consecutive records** such that the **first one** has “Amount” **lower** than 3,000 and the **second one** has “Amount” **higher or equal** to 3,000. Then, **all the records** from the beginning of the **record-list up to, but not including**, the one with “Amount” higher than 3,000 are **exactly the ones** that satisfy the “**WHERE**” expression.

The principle of indexing is exactly the same as the one of ordering the elements in any directory (words in a paper dictionary, terms in an encyclopedia, files in a file cabinet, flights in an airport bill-board, etc.). Because the elements (words, files, flights, ...) are **ordered** according to some criterion (alphabetically, by flight time, by flight destination city, ...), you can find the one you want in a **fast way**. Imagine how much time it would take you to find the word you want in a paper dictionary if it had an unknown ordering of words!!!

As you have noticed from the examples that I have just shown, **indexing and binary search** require **much fewer** record fetching and value checking operations than the brute-force approach of looking through 1,000,000 records **one by one**. If we call “N” the number of elements in the list, **binary search** only requires in the order of **log<sub>2</sub>(N) lookups**, while a normal (i.e., sequential) search over a non-indexed list requires an average of **N/2 lookups**. For a list of N=1,000,000 records, N/2 is 500,000 while **log<sub>2</sub>(N)** is **only** around **20**! As you may see, the advantage in processing time is enormous: from doing record fetching and lookup over **500,000** records **down to** only **20** records!! The longer the list, the largest the advantage of indexing plus binary search over sequential search.

But probably you may think the following:

*Yeah, great, but you need to know in advance that I am going to do a search on the field “Amount”. What if I use a “**WHERE**” clause over “Invoice\_Date”, or over some other field? Then, having the list ordered by “Amount” would not help a bit, right??*

You are right, **but**, the great thing about indexing is that you may have each record-list **ordered under many field names at the same time**. This is so because indexing **does not change the order of the records** in the record-list. The records are **statically stored** in some position of the computer storage, and the system does not move them around (it would be too costly in terms of processor time). What the system does to order a record-list is to create an **index**, this is, an **ordered list of record pointers**. Recall that a record pointer is an **internal** storage address of the database that allows the system to locate each record stored in the database. Therefore, the system can build and store **several indexes**, this is, **several ordered lists of record pointers**. **Each index** is ordered following the criterion of the values of **one specific field name**. If a record-list has an index on a given field name, then we say the record-list is **indexed on that field name**.

Let me try to show all of this with an example. Let us consider the following Table of invoice records:

R_POINTER	Invoice_Date	Invoice_Num	Amount	Invoice_VAT	Description	Customer
103==>	03-ene-15	501	457,56	96,09	Transceiver	Cisco
104==>	04-sep-15	503	1.435,34	301,42	Wifi Access Point	NEC
105==>	08-mar-15	502	747,67	157,01	Wifi Access Point	Telefonica
106==>	08-jul-15	504	1.335,00	280,35		Tata
107==>	07-ene-16	505	1.892,90	397,51	IP Switch	Huawei
108==>	26-oct-16	506	4.125,89	866,44	2000 SIM cards	Telefonica
109==>	26-feb-16	507	144,70	30,39		Tata
110==>	13-ene-17	508	2.690,55	565,02	131 cans	Huawei
111==>	18-dic-17	509	5.160,00	1.083,60	IP Switch	Tata
112==>	15-feb-17	510	2.243,01	471,03	Consulting Services	Cisco
113==>	20-ene-18	512	1.058,50	222,29	IP Switch	NEC
114==>	05-oct-18	511	1.695,56	356,07	1678 SIM cards	Telefonica
115==>	05-feb-18	513	6.540,04	1.373,41	Wifi Access Point	NEC

The first column shown (called “R\_POINTER”) **is not a field name** of the Table. Rather, it represents the “**record pointer**”, which is the internal identifier used by the database system to locate each record in the system storage. If we want to index this Table under **each and every** of its field names, then the database system will build the following six indexes (i.e., six ordered lists of record pointers), each index corresponding to the

ordered values of each of the record's field names:

Index on Invoice_Date	Index on Invoice_Num	Index on Amount	Index on Invoice_VAT	Index on Description	Index on Customer
103==>	103==>	109==>	109==>	106==>	112==>
105==>	105==>	103==>	103==>	109==>	103==>
106==>	104==>	105==>	105==>	110==>	110==>
104==>	106==>	113==>	113==>	114==>	107==>
107==>	107==>	106==>	106==>	108==>	113==>
109==>	108==>	104==>	104==>	112==>	104==>
108==>	109==>	114==>	114==>	107==>	115==>
110==>	110==>	107==>	107==>	111==>	106==>
112==>	111==>	112==>	112==>	113==>	109==>
111==>	112==>	110==>	110==>	103==>	111==>
113==>	114==>	108==>	108==>	104==>	114==>
115==>	113==>	111==>	111==>	105==>	108==>
114==>	115==>	115==>	115==>	115==>	105==>

As you may see, **one** given **record-list** can have **several indexes** associated to it, and this is actually **quite frequent** in database Tables.

Notice that some of the field value-lists in this example have duplicate values and/or **Nulls**.

There is no problem with this, because **indexing works** in the presence of duplicate values and/or **Nulls**. When a field value-list has duplicate values, the index lists each **group** of duplicate values in the correct order **in respect** to the other values, and the duplicate values do not follow any specific order among themselves. Regarding **Nulls**, for the purpose of indexing they are considered as the same “unknown” value, and they are placed the first ones (in ascending order) in the index. To clarify this, I have marked with the same color the groups of duplicate field values (or **Nulls**) under the field name “Description”. You may see that each group of duplicate values is correctly ordered in respect to the other values.

The relative order of the records **within** each group of duplicate values can be whatever: the example above could order the pointers **with the same color** in a different way and the index would still be totally correct.

### C.8.2 Can I create one index over a list of field names (called a composite index)?

Yes, we can! A **composite** index is **one** index created with the ordering criterion of **a list of fields** instead of the ordering criterion of just one field, as you have seen up to now.

The way to build a **composite** index is by starting to order the record pointers on the values of the **first** field name. Each **group of record pointers** corresponding to **the same value** in the first field is then **ordered** on the values of the **second** field in the list of fields. Each **group** of record pointers that have the same value A in the first field and also the same value B in the second field is then ordered on the values of the **third** field, and so on.

When configuring a **composite** index, you can choose for **each** field in the list of fields

if you want ascending or descending as an ordering criterion of the values of that field.

If you created a **composite** index on the list of field names “Description”, “Customer” and “Amount”, all in ascending order, you would get the following result:

T_Invoices_Indexing						
R_POINTER	Invoice_Date	Invoice_Num	Amount	Invoice_VAT	Description	Customer
109==>	26-feb-16	507,00	144,70	30,39		Tata
106==>	08-jul-15	504,00	1.335,00	280,35		Tata
110==>	13-ene-17	508,00	2.690,55	565,02	131 cans	Huawei
114==>	05-oct-18	511,00	1.695,56	356,07	1678 SIM cards	Telefonica
108==>	26-oct-16	506,00	4.125,89	866,44	2000 SIM cards	Telefonica
112==>	15-feb-17	510,00	2.243,01	471,03	Consulting Services	Cisco
107==>	07-ene-16	505,00	1.892,90	397,51	IP Switch	Huawei
113==>	20-ene-18	512,00	1.058,50	222,29	IP Switch	NEC
111==>	18-dic-17	509,00	5.160,00	1.083,60	IP Switch	Tata
103==>	03-ene-15	501,00	457,56	96,09	Transceiver	Cisco
104==>	04-sep-15	503,00	1.435,34	301,42	Wifi Access Point	NEC
115==>	05-feb-18	513,00	6.540,04	1.373,41	Wifi Access Point	NEC
105==>	08-mar-15	502,00	747,67	157,01	Wifi Access Point	Telefonica

You may check that the list of record pointers (i.e., the **index**) is ordered first on ascending value of “Description”, then on ascending value of “Customer” and finally on ascending value of “Amount”.

Although the relevance of this fact is usually low, I want to point out that the fields that make a composite index are stated in **order**, and the **order** you choose has **some impact** on the performance of the index. As I stated slightly above, the records in the list are ordered on the first place according to the values of the **first** field in the index, the ones with the same value in the first field are then ordered among themselves according to the value of the second field in the index and so on. I will not go into the detail of why this impacts performance, and will only indicate that to improve performance, you should put first the fields with less duplicate values in the record-list (i.e., the ones that are more “unique”) and last the fields with more duplicate values.

Notice further that, like it happens with a simple index, a **composite** index may also be built over a record-list that has duplicate **value arrays** in the fields belonging to the index. When I say duplicate “**value array**” I mean that the values of two records, when comparing, field name by field name, the values of the fields in the index, have the same values in all the field pairs of fields belonging to the index. This is, two or more records have the same **value array** in the fields belonging to the index. Like in the simple index case, this is not a substantial problem and indexing works fine over composite indexes in the presence of duplicate value arrays: it is just less efficient (the more duplicates, the less efficient).

If you want to know how to actually configure a composite index with MS-Access, you may click “[D.7 How do I add simple and/or composite index\(es\) to a Table?](#)”.

### C.8.3 What different types of indexes are there?

You may click:

- “C.8.3.1 What are simple indexes and composite indexes?”
- “C.8.3.2 What are indexes with duplicate values and without duplicate values?”
- “C.8.3.3 What are indexes that ignore Nulls and not ignore Nulls?”
- “C.8.3.4 What are indexes with Nulls and without Nulls?”
- “C.8.3.5 What is an index without duplicate values and without Nulls?”

#### C.8.3.1 What are simple indexes and composite indexes?

You can configure **simple** indexes and **composite** indexes. I described **composite** indexes in “C.8.2 Can I create one index over a list of field names (called a composite index)?”.

A **composite** index is built over one **ordered list of fields**, while the simple index is built over only **one** field. There is **no conceptual difference** between a **simple** and **composite** index: a **simple** index is just a **particular case** of the more general concept of **composite** index. A **simple** index is the particular case where there is only one field in the index field list. However, it is usual practice to distinguish both and this is why a specific name has been assigned to each.

Because a **simple** index is just a particular case of a **composite** index, I will do all my explanations about “**indexes**” in a generic way. I will only make a distinction between a **simple** and a **composite** index if it is required for some reason.

#### C.8.3.2 What are indexes with duplicate values and without duplicate values?

As I explained in “C.8.1 Why is indexing useful?”, indexing works and serves to improve performance. Performance is improved even in the presence of duplicate **value arrays** in the index fields across the records in the Table.

Therefore, it makes sense to configure indexes even if there are duplicate value arrays.

However, it also makes sense to configure indexes that **prevent** the existence of duplicate **value arrays** in the fields of the index. I will call this type of index an index **without duplicate values**. When this type of index is configured in a Table, the system prevents inputting in the Table any new record that has a **duplicate value array** in the fields of the index in respect to **every other record** already existing in the Table. This type of index guarantees that **each and every** record in the Table will have a **different value array** in the Table fields that are included in the index. Just to clarify, when I say same “**value array**” this means that the values of the fields are the same **comparing fieldname by fieldname** between two different records, considering the fields belonging to the index.

Notice that guaranteeing that the index **value arrays** are **unique** does **not** guarantee that there will not be duplicate **records**. The reason for this is the possible presence of **Nulls** in the fields belonging to the index. Because **Null** is not a value, two **Nulls** in the same field in two different records **will be considered as different values** for the purpose of accepting a newly input record into the Table. This causes that even if you configure an **index without duplicate values**, it is possible to have duplicate records in the Table.



### **C.8.3.3 What are indexes that ignore Nulls and not ignore Nulls?**

As I have shown in “*C.8.1 Why is indexing useful?*”, indexing works and serves to improve performance even in the presence of **Nulls** in the index fields. Therefore, it makes sense to configure indexes that **include** in the index the records that have **Nulls** in the index fields, across the records of the Table. This is called in MS-Access “**not ignore Nulls**”.

However, it also makes sense to configure indexes that **exclude** from the index the records that have **Null** in one or more index fields. Excluding from the index the records with **Nulls** in one or more index fields will make the index have a better performance for the records that do not have **Nulls**. This is called in MS-Access “**ignore Nulls**”

I want to highlight that it is **very different** to configure an index to **ignore Nulls** (i.e., not including in the index the records that have one or more **Nulls** in the index fields), than to configure the index fields as **without Nulls** (i.e., configuring all the fields in the index as “**Required=Yes**” and making sure the fields do not contain any pre-existing **Null**). Configuring an index to **ignore Nulls** just makes the index slightly more efficient, but it **does not** guarantee the absence of **Nulls** in the index fields of your Table records. However, configuring **all** the index fields as **without Nulls** actually enforces that there are no **Nulls** in the index fields. Notice that in case there are **Nulls** in a Table field when you configure it as “**Required=Yes**”, MS-Access will issue a warning message, but the **Nulls** will stay there. If you want the field value-list to contain no **Nulls**, you should **manually remove** from the field value-list all its pre-existing **Nulls**.

### **C.8.3.4 What are indexes with Nulls and without Nulls?**

An **index without Nulls** is when **all** the **field value-lists** in an index **do not** contain any **Null**. Otherwise, I will call it an index **with Nulls**. To configure a Table field as **without Nulls** you have to set its **Required** property to “**Yes**” (click *D.5.1.7*), and in case you get a warning that the field contains **Nulls**, you should **manually remove all the Nulls from the field**.

Notice that in MS-Access the property “**Required**” is **not** part of the configuration of the index itself, and rather it is a configuration of each individual Table **field**. Therefore, configuring a given Table **field** as **with Nulls** or **without Nulls** will first affect **the field itself**, because it cannot have **Nulls** across all Table records, and second, this will affect all the Table indexes that include the field.

I want to highlight that it is **very different** to configure an index to **ignore Nulls** (i.e., not including in the index the records that have one or more **Nulls** in the index fields), than to configure the index fields as **without Nulls** (i.e., configuring all the fields in the index as “**Required=Yes**” and making sure that the index fields do not contain any pre-existing **Null**). Configuring an index to **ignore Nulls** just makes the index slightly more efficient, but it **does not** guarantee the absence of **Nulls** in the index fields of your Table records. However, configuring **all** the index fields as **without Nulls** actually enforces that there are no **Nulls** in the index fields. Recall that in case there are **Nulls** in a Table field when you configure it as “**Required=Yes**”, MS-Access will issue a warning message and you should manually remove from the field all its pre-existing **Nulls**.

### **C.8.3.5 What is an index without duplicate values and without Nulls?**

An index **without duplicate values** and **without Nulls** is an index configured as

**without duplicate values** (click *C.8.3.2* and to configure *D.7.1* and *D.7.2*) and where **all its fields** are configured as **without Nulls** (click *C.8.3.4* and to configure *D.5.1.7*).

This should not be mistaken with an index configured as **without duplicate values** and **ignore Nulls** (click *C.8.3.3* and to configure *D.7.1* and *D.7.2*), which is a very different thing.

An index **without duplicate values** and **without Nulls** has **several** important properties:

- It **prevents duplicate records** in the Table.  
If you want more detail on this, you may click “*C.9 How do I prevent duplicate field values and duplicate records?*”.
- The index fields can be configured in MS-Access as the **Primary Key**.  
If you want more detail on this, you may click “*C.10 What are the Table Key(s) and how should I handle them?*”.
- The index fields can be the **master fields** in a **Relationship with referential integrity**.  
If you want more detail on this, you may click “*C.11.1 What is a Relationship with referential integrity?*”.
- The index fields can be used in a **many-to-many Relationship**.  
If you want more detail on this, you may click “*C.11.4 What is a many-to-many Relationship?*”.
- Any superset of the index fields can be the **slave fields** in a **one-to-one Relationship**.  
If you want more detail on this, you may click “*C.11.2 What are “one-to-many” and “one-to-one” Relationships?*”.

If you want to know more about the interactions between **Nulls**, duplicates, indexing and **Key** fields, you may click “*K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?*”.

### **C.8.4 What indexes can I configure in a given Table?**

Some important remarks that you should know about configuring indexes in your Tables are the following:

- Remind that configuring **indexing over one individual field name** (i.e., a **simple** index) is just a **particular case** of configuring indexing over a **list of field names** (i.e., a **composite** index). The particular case is the one where **the list only contains one field name**.
- You can configure **several simple indexes** and/or **composite indexes** in the **same Table**.
- A given Table field may **belong to several** composite **indexes**, and it can also be a **simple** index in itself, all at the same time.
- You can configure, on a given Table, some indexes **with duplicate values** (only to improve performance) and other indexes **without duplicate values** (to **also** restrict the values of records in the Table).
- You can configure each index in the **same Table** to **include or not** (i.e., not ignore or ignore) the records that have one or more **Nulls** in the index fields. Configuring

an index to **not include** records with **Nulls** just makes the index slightly more efficient, but it **does not** prevent **Nulls** in the corresponding fields of your Table records.

- It is **very different** to configure an index to **ignore Nulls** (i.e., not including in the index the records that have one or more **Nulls** in the index fields), than to configure the index fields as **without Nulls** (i.e., configuring all the fields in the index as “**Required=Yes**” and making sure that the fields do not contain any pre-existing **Null**). Configuring an index to **ignore Nulls** just makes the index slightly more efficient, but it **does not** guarantee the absence of **Nulls** in the index fields of your Table records. However, configuring **all** the index fields as **without Nulls** (i.e., configuring all the fields in the index as “**Required=Yes**” and making sure that the fields do not contain any pre-existing **Null**) actually enforces that there are no **Nulls** in the index fields. Recall that in case there are **Nulls** in a Table field when you configure it as “**Required=Yes**”, MS-Access will issue a warning message, but the **Nulls** will stay in the Table: if you do not want **Nulls**, you should manually remove from the field all its pre-existing **Nulls**.
- Linking with the previous point, it is very different to configure an index **without duplicate values** and **ignore Nulls**, than to configure an index **without duplicate values and without Nulls**. The latter index has very valuable properties (click C.8.3.5), while the former index is just an index without duplicate values where records with **Null** in one (or more) index field(s) are not included in the index.
- Combining the configuration of **duplicate values**, “**Required**” (i.e., without **Nulls**) and ignoring **Nulls**, you can configure in the same Table or in different Tables:
  - Indexes **with duplicate values, with Nulls and not ignore Nulls**
  - Indexes **with duplicate values, with Nulls and ignore Nulls**
  - Indexes **with duplicate values and without Nulls**
  - Indexes **without duplicate values, with Nulls and not ignore Nulls**
  - Indexes **without duplicate values, with Nulls and ignore Nulls**
  - Indexes **without duplicate values and without Nulls**
- It is **very different** to configure a **composite index** over a given list of Table field names than configuring a **simple index** over each and every **individual** field in the list. The **composite index** will improve performance when expressing Query conditions over the **whole list of fields** (e.g., in Relationships), while each **simple index** will improve performance when expressing Query conditions over **each individual field**.
- It is **very different** to configure a **composite index without duplicate values** over a given list of Table field names than to configure a **simple index without duplicate values** over each and every **individual** field in the list. Configuring a **simple index** over each and every **individual** field name in the list is a much more restrictive condition, because it implies there should be no duplicate values in each of the individually indexed fields, while the composite index only prevents duplicate **value arrays** over all the index fields.
- It is **very different** to configure a **composite index without duplicate values and without Nulls** values over a given list of Table field names than to configure a **simple index without duplicate values and without Nulls** over each and every **individual** field in the list. In the **second** case, you can configure **each and every**

of the index fields as the **Primary (simple) Key**, while in the first case you cannot.

- If you configure **indexing without duplicate values** over a **given group** of Table field name(s) called **{Field\_group}**, you can **also** configure indexing without duplicate values over **any list of field names that includes {Field\_group}** and the no-duplicate-value condition is **guaranteed for sure**. This is so because just **one field name** that has no duplicate values, implies that any group of field names that includes that field will have no duplicate values: the values of this individual field **will be different for sure in all the Table records, regardless of the values of the other field names in the list**.

### **C.8.5 Why should I use indexing in my database?**

Configuring indexes in your Tables is **very useful**, and it should be done in **almost every Table**.

Having **one or more** indexes (with or without duplicate values and with or without **Nulls**) in a Table **serves a very relevant purpose**:

- **Largely improving the performance of your database**  
You may click “*C.8.5.1 Why indexing improves performance?*”.

Having **at least one** index **without duplicate values and without Nulls** in a Table **serves an extremely relevant purpose**:

- **Preventing duplicate records in the Table**  
You may click “*C.8.5.2 Why indexing prevents duplicate records in a Table?*”.

Having **one or more** indexes **without duplicate values and without Nulls** in a Table **serves a very relevant purpose**:

- **Establishing Relationships with referential integrity between Tables**  
You may click “*C.8.5.3 Why indexing allows establishing Relationships between Tables?*”.

#### **C.8.5.1 Why indexing improves performance?**

If my explanation in “*C.8.1 Why is indexing useful?*” was so bad that you do not see how/why indexing works, you should not worry a bit: just take my word for it (hey, I am a Professor!).

The only relevant point is that you **should configure indexing** in **every Table field name**, or list of field names, over which it is **likely** that you **will establish** some Query condition (e.g., a “**WHERE**” or “**ON**” expression). The reason is that this will make your Queries **much faster**. If that field or list of fields can have duplicates or **Nulls**, you should configure the index to allow duplicates or not, and ignore **Nulls** or not, as it corresponds according to other criteria.

#### **C.8.5.2 Why indexing prevents duplicate records in a Table?**

Duplicate records in Tables (click “*C.7.1 What are duplicate records?*”) create significant risks of incorrect interpretation of the records and it is difficult to find a sound reason why you would need them. Preventing duplicate records in Tables is done automatically by the database if you define **at least one** index **without duplicate values and without Nulls**.

I **strongly recommend** you prevent duplicate records in (almost) **all** your Tables.

Remind that each field, or group of fields, having an index **without duplicate values and without Nulls** can be configured in MS-Access as the **Primary Key** of the Table. If you want to know more about the concept of Table **Key**, you may click “[C.10 What are the Table Key\(s\) and how should I handle them?](#)”.

### **C.8.5.3 Why indexing allows establishing Relationships between Tables?**

I have not yet explained Relationships, but I will do it in “[C.11 What is a Relationship?](#)” further down. You can click [C.11](#), read it, and then come back, or rather, continue reading here just believing what I say.

You can only create a **Relationship with referential integrity** (i.e., a one-to-many or a one-to-one Relationship) if the **master** field(s) have **an associated index without duplicate values and without Nulls**. Notice that it is **not** enough that **some** of the **master** fields have such an index: what is required is that **exactly** the master fields have **an associated index without duplicate values and without Nulls**. If a Table **does not have any index without duplicate values and without Nulls**, it is **impossible** to establish a Relationship with referential integrity where that Table is the **master** Table (i.e., **from** that Table **to** any other Table).

You can only create a **one-to-one** Relationship if the **slave** the fields (or any **subset** of the) have **an index without duplicate values and without Nulls**. If the **slave** Table **does not have any such index**, it is **impossible** to establish a **one-to-one** Relationship to it.

You can only create a **many-to-many Relationship** between several Tables if each and every Tables has **an index without duplicate values and without Nulls** with the **same number** of fields and where the list of field type and size of the index fields of all the Tables is the same. Otherwise, it is **impossible** to establish a many-to-many Relationship between that Tables.

You will certainly want to define Relationships in your database, so you should first define **at least one** index **without duplicate values and without Nulls** in (almost) every Table, each time you create it.

## **C.9 How do I prevent duplicate field values and duplicate records?**

You may click:

- “[C.9.1 How I do I prevent duplicate field values over a group of field name\(s\) in my Tables?](#)”
- “[C.9.2 How I do I prevent duplicate records in my Tables?](#)”
- “[C.9.3 How do I prevent duplicate records in my Query's record-lists?](#)”

### **C.9.1 How I do I prevent duplicate field values over a group of field name(s) in my Tables?**

The way the system prevents **duplicate values** over a group of field names in a Table is by checking if any new record that the user wants to insert in the Table is a duplicate value over that group of fields. This implies that every time you want to insert a new

record, the system must search **the whole Table**, to check if the new record is a duplicate value over the group of fields. How can we do this checking in a fast way? The answer is simple: build an **index without duplicate values and without Nulls** over that **group of field names!!** This will allow the system to automatically check in a very fast way if the field values of the new record that is being inserted are duplicate values or not, and therefore, the system can decide if the new record can be inserted or not.

Therefore, if you want to configure the system to prevent duplicate values in a Table over a group of field names, you **must** configure **indexing without duplicate values** over that group of fields in that Table.

If you configure indexing **without duplicate values** over a group of field names, the database system **will not allow you** to input a **new** record with **the same values in the group of field names** as the ones in **another record already** existing in the Table.

Notice that, in order to prevent **duplicate records**, you have to configure all the fields in the index as **without Nulls in addition** to configuring the index as **without duplicate values**. This is so because an index configured as without duplicate values **does not** consider two (or more) **Nulls** as “the same value”, because indexing considers **Nulls** as “unknown”, and therefore, they can be the same value or not. Therefore, if you configure an index without duplicates but allowing **Nulls**, the database system will allow you to insert new records with **Null** in one (or more) of the fields of the index, even if there is already another record **with the same values in the other fields** in the group. Even if you already have records having **Null** in the same field(s) as the new record you are inserting, the system will allow you to insert the new record.

Therefore, if you only **prevent duplicates over field values** (i.e., by configuring an index *without duplicate values*) this will **not** prevent **duplicate records**, because duplicate records can happen because of having **Null** in some fields.

If you want to know more about how to configure indexing in MS-Access, you may click “[D.7 How do I add simple and/or composite index\(es\) to a Table?](#)”.

### **C.9.2 How I do I prevent duplicate records in my Tables?**

You can make the system **prevent duplicate records** by **preventing duplicate values on a group of field names, plus, preventing Nulls in each and every field name** in the group of field names. This is done by **configuring** the system to enforce **indexing without duplicate values** over a group of field names, **plus**, configuring **each and every field name in the group** as **without Nulls** (i.e., configuring it as “**Required=Yes**”).

Notice that if there are no duplicate values over a group of field names, and also, there are no **Nulls** in the group fields, then it is **impossible** to have a duplicate record (unless there were pre-existing values).

If you configure **indexing without duplicate values** over a group of field names, **plus**, configuring each and every field name in the group as **without Nulls** (i.e., as “**Required=Yes**”), MS-Access **will not allow you** to input a **new** record with **the same values, or Null, in the group of field names** as the ones in **another record already** existing in the Table.

Some important remarks about configuring **indexing without duplicate values** over one groups of field names, **plus**, configuring all the field names in the group as **without**

**Nulls** (i.e., as “**Required=Yes**”):

- You can configure **indexing without duplicate values and without Nulls** over **several** groups of field names: **each group of fields names** just requires **its own specific composite index and each individual field configured without Nulls**.
- Configuring **indexing without duplicate values and without Nulls** over **one individual field name** is just a **particular case** of a group of field names where **the group only contains one field name**.
- It is **not the same** to configure **indexing without duplicate values and without Nulls** over a **composite** group of field names than configuring **indexing without duplicate values and without Nulls** over each and every **individual** field name in the group. Configuring **indexing without duplicate values and without Nulls** over each and every **individual** field name in the group is a much more restrictive condition.

To configure **indexing without duplicate values and without Nulls** over **composite** fields (or over individual fields), you just configure indexing without duplicate values as indicated in “*C.9.1 How I do I prevent duplicate field values over a group of field name(s) in my Tables?*”, and in addition, you configure each and every Table field as **without Nulls**.

Remind that to configure a Table field as **without Nulls** you have to open the Table in “**Design View**” and click on the field you want to configure. Find the row “**Required**” in the field properties, placed at the bottom of the “Table pane”, in the tab “**General**”. Right-click on the rightmost side of the row and click on “**Yes**” from the drop-down menu. This **guarantees** that MS-Access does not allow you to enter any **NULL** into this field (but notice that pre-existing **Nulls** will stay in the Table, unless you manually remove them).

If you want to know more about how to configure your indexes, you may click:

- “*D.5.1.7 What is the “Required” Table field property?*”
- “*D.5.1.8 What is the “Indexed” Table field property?*”
- “*D.6 How do I configure the Primary Key field(s) of a Table?*”
- “*D.7 How do I add simple and/or composite index(es) to a Table?*”

If you want to know more about the interactions between **indexing**, **Nulls** and **Key** fields, you may click:

- “*K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?*”

### **C.9.3 How do I prevent duplicate records in my Query’s record-lists?**

Record-lists produced by **Queries**, in particular during intermediate processing, **may have** duplicate records. Even if **all** the database Tables are duplicate-free, the Query processing **may generate** duplicate records.

If you want to prevent duplicates in the **output** record-list of a **Select** operation, you only need to add the optional clause “**DISTINCT**”, and all duplicate records will be removed, **leaving one**, and only one, record **from each group** of duplicate records.



If you want to prevent duplicates in the **output** record-list of a **Union** operation, you only need to use the operator “**UNION**”, instead of “**UNION ALL**”. This will remove all duplicate records from the **output** record-list, **leaving one**, and only one, record **from each group** of duplicate records. Notice this will remove duplicate records existing in **each** of both **input** record-lists, even if they are not duplicates in respect to the other record-list.

Notice that using “**DISTINCT**” or “**UNION**” (vs. not using “**DISTINCT**” or using “**UNION ALL**”) will make your Queries **slower** because the system has to check for duplicate records, and remove them if found, before producing the output record-list.

If you want to know more about this, you may click:

- “*F.7.11 What is the “DISTINCT” clause of a Select?*”
- “*F.9.1 What are the Union operators?*”

## **C.10 What are the Table Key(s) and how should I handle them?**

You may click:

- “*C.10.1 What is the Primary Key of a Table?*”
- “*C.10.2 What is a simple Key and a composite Key?*”
- “*C.10.3 What are the candidate Keys of a Table?*”

If you want to **configure** the **Primary Key** in MS-Access, you may click:

- “*D.6 How do I configure the Primary Key field(s) of a Table?*”.

### **C.10.1 What is the Primary Key of a Table?**

The **Primary Key** of a Table is **one, and only one, minimal** group of field names used by the system to **jointly identify** the **type of entities** represented in this Table, and also, to **jointly identify** the **record** corresponding to **each entity** in the Table. The **Primary Key** may be composed of **one or more** field names.

The **group** of field(s) of **the Primary Key** must satisfy the following **four** necessary and sufficient conditions:

- **No Primary Key field can contain Null.**
- The combination of values of the **Primary Key field(s)** must be **different** in each and every Table record.
- **No subset** of the **Primary Key fields** may also identify each record (entity) in the Table. This condition is called being “**minimal**”.
- A Table can have **one, and only one, Primary Key.**

MS-Access **enforces** the two first conditions above by automatically configuring an index without duplicate values and without **Nulls** for the **Primary Key** field(s). MS-Access also enforces the fourth condition, and if you configure a second **Primary Key**, the previously existing one will be cleared. However, MS-Access **does not** enforce the third condition (being minimal): it is therefore possible to define a **Primary Key** for

**any superset** of the **Primary Key field(s)**. This allows more design flexibility but may also lead to worse database designs.

Linking with the sections on **duplicates** (C.7, C.9) and **indexing** (C.8), the **Primary Key field(s) must have an index without duplicate values and without Nulls**. MS-Access will automatically create this index when you configure the **Primary Key field(s)**.

The term “**Primary Key**” of the Table is frequently shortened to just saying the “**Key**” of the Table.

Let us see one example of **Primary Key**, based on the following Table that lists employees:

T_Employees				
Social_Security_Num	Given Name	Middle Name	Family Name	Birth_Date
34562444	Arturo		Azcorra	09-dic-65
52674883	John	William	Doe	13-dic-75
64537727	Silvia		Johnson	13-feb-79

It makes all the sense to have “Social\_Security\_Num” as the **Primary Key** (i.e., the field name that identifies each entity in the Table), while “Given\_name”, “Middle\_name”, “Family\_name” and “Birth\_Date” are fields that provide information about each entity in the Table. Each entity is an employee and is uniquely identified by his/her Social Security number. His/her given name, family name and birth date are his/her attributes and are registered in other fields in his/her **specific and unique** record. Notice that **in this case the group of field names that constitute the Primary Key of the Table is only one field name**.

When you design a database, you create Tables with many purposes. Some Tables will serve the purpose of listing different types of entities (invoices, pay slips, cities, engine parts, car models, addresses, ...) that are relevant for you. **Each** of these Tables contains a listing of **all entities** of a **given type**. You can have the Table of invoices, or the Table of engine parts, or the Table of Capital Cities. In each of these Tables, some fields serve to identify each particular entity (e.g., a specific invoice or engine part), and the other fields provide data over the entity (e.g., the invoice date, the invoice amount, ...). The **Primary Key field(s)** are the ones used by the database system to identify the entity itself. This is what I have just shown in the example of the Table of employee data.

If you want to know more about the interactions between indexing, **Nulls** and **Primary Key field(s)**, you may click “K.2.3 *What are the interactions between Nulls, duplicates, indexing, and Key field(s)?*”.

### C.10.2 What is a simple Key and a composite Key?

A **Key** may be composed of **one** or **more** field names. There is **no conceptual difference** between a **Key** being **one** or **more** field names. However, for practical purposes it is convenient to distinguish both cases, and this is why most books use a specific **term** for each case:

- A “**simple**” **Key** is the one composed of **only one field name**.
- A “**composite**” **Key** is the one composed of **more than one field name**.

### C.10.3 What are the candidate Keys of a Table?

A **candidate Key** of a Table is **any minimal group of field names** that **could be used** by the system to **jointly identify** the **type of entities** represented in this Table, and also, to **jointly identify** the **record** corresponding to **each entity** in the Table. Each **candidate Key** may be composed of **one or more** field names. **The Primary Key** is therefore **one** of the **candidate Keys** of the Table, and **each candidate Key** if a feasible **option** to be the **Primary Key**.

The **field(s)** of **each candidate Key** must satisfy the following **three** necessary and sufficient conditions:

- **No candidate Key field can contain Null.**
- The combination of values of the **candidate Key field(s)** must be **different** in each and every Table record.
- **No subset** of the **fields** may also identify each record (entity) in the Table. This condition is called being “**minimal**”.

If you want to configure in MS-Access a set of field(s) as a **candidate Key**, you will have to enforce the conditions above. The way to do it is by configuring an **index without duplicate values and without nulls** for the field(s) you want to become a **candidate Key**. MS-Access **does not require you to enforce** the third condition (being minimal): it is therefore possible to define a **Primary Key** for **any superset** of the **Primary Key field(s)**. This allows more design flexibility but may also lead to worse database designs.

Linking with the sections on **duplicates** (C.7, C.9) and **indexing** (C.8), you **may configure an index without duplicate values and without Nulls for the field(s) of any candidate Key**.

If you want to know more about the interactions between indexing, **Nulls** and **Primary Key field(s)**, you may click “*K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?*”.

## C.11 What is a Relationship?

Creating a **Relationship** is **relating an ordered list** of Table **field names** (the **master fields**) with another **ordered list** of Table **field names** (the **slave fields**) such that the **values** of the **slave fields** in **any slave** record **must be equal** to the values of the **master fields** in **one of the existing master** records. The Table of the **master fields** is the **master Table** and the Table of the **slave fields** is the **slave Table**.

Let us see this with an example. I can create a Relationship from the **master field** “City” from the **master Table** “T\_Subsidiary\_Sites” to the **slave field** “Capital” from the **slave Table** “T\_Capital\_Cities”. This means that the value of the field “T\_Capital\_Cities.Capital” **must take the value** of the field “City” from one of the **existing** records of the Table “T\_Subsidiary\_Sites”. If you now go over the values of the

following two Tables

T_Subsiary_Sites	
City	Country
Beijing	China
Bilbao	Spain
Brasilia	Brazil
Buenos Aires	Argentina
Sidney	Australia
Madrid	Spain
Sao Paulo	Brazil
Stuttgart	Germany
Washington	United States

T_Capital_Cities		
Capital	State_Province	Country
Beijing	No_data	China
Brasilia	No_data	Brazil
Buenos Aires	No_data	Argentina
Madrid	Madrid	Spain
Washington	District of Columbia	United States

you can check that **all** the values of “City” in the **slave records** (from “T\_Capital\_Cities”) **satisfy** the Relationship you created in respect to the values of “Capital” in the **master records** (from “T\_Subsiary\_Sites”).

However, if you now entered a record in the **slave** Table “T\_Capital\_Cities” with the “Capital” value “Moskow”, then you would be **violating** the Relationship, because there is **no record** in the **master** Table “T\_Subsiary\_Sites” with the value “Moskow” in its field “City”.

A Relationship is a **very simple concept**, but, it has **surprisingly useful and complex applications**.

I will now make a few clarifications over Relationships.

If a given Relationship involves **more than one field**, the relation between **master** fields and **slave** fields is defined **by field pairs** (i.e., **each specific master field is related to one specific slave field**). For example, you can establish one Relationship between the **master** Table “T\_Subsiary\_Sites” and the **slave** Table “T\_Capital\_Cities” in which “City” is the **master field** of “Capital” **and** “Country” (from “T\_Subsiary\_Sites”) is **the master** field of “Country” (from “T\_Capital\_Cities”).

Relationships are **not commutative** (i.e., they are directional). Notice that it is **very different to say**:

Each **slave field** “Capital” from “T\_Capital\_Cities” **must take** a value from the **master field** “City” in **one existing record** of “T\_Subsiary\_Sites”.

from saying:

Each **slave field** “City” from “T\_Subsiary\_Sites” **must take** a value from the **master field** “Capital” in **one existing record** of “T\_Capital\_Cities”.

As you may see, the **values** of the **slave** field(s) in **all the records** from the **slave** Table are a **subset** of the **values** of the **master** field(s) in **all the records** from the **master** Table. This clearly shows that a Relationship is **not commutative**.

Notice that a given Table may be involved in **many** Relationships with **many** other

Tables, but **any given Relationship** is **always** defined between **two, and only two, Tables**.

It is possible (although very unusual) to establish a Relationship between fields of the same Table. If you do this, the **master** Table and the **slave** Table is the **same** Table.

It is **different** to establish **one** Relationship over **two** (or more) field pairs, than establishing **two** (or more) **different** Relationships each of them over **one** field pair. I will show this difference with the last example above. Notice that you just created **one Relationship** between the **two** master fields “City” and “Country” in respect to the two slave fields “Capital” and “Country”. This means that “Capital” and “Country” can only take values from **an existing record** in the master Table “T\_Subsidiary\_Sites”. If this is fulfilled, the records in “T\_Capital\_Cities” will be correct, because all of them will have a correct value in the “Capital” and “Country” coming from **some record** of “T\_Subsidiary\_Sites”. Now, if you create **instead two Relationships**, one Relationship between “City” and “Capital” and **another** Relationship between “Country” and “Country”, the situation is quite different. Now, the field “Capital” can take **any existing value** from the field value-list “City”, and the field “Country” (from “T\_Capital\_Cities”) can take **any existing value** from the field value-list “Country” (from “T\_Subsidiary\_Sites”). This means that [Beijing, Spain] would be a valid record in “T\_Capital\_Cities” because “Beijing” is a valid value (it exists in its corresponding master field value-list “City” in “T\_Subsidiary\_Sites”), and “Spain” is also a valid value (it exists in its corresponding master field value-list “Country” in “T\_Subsidiary\_Sites”). However, it is **pretty obvious** that the record [Beijing, Spain] is **wrong**. You can now see that establishing **one Relationship** between **two field pairs**, and establishing **two Relationships**, each between **one field pair** is clearly different.

Let me summarize the characteristics of Relationships that I have presented above:

- The **values** of the **slave fields** in **every slave record must** be the same as the **values** of the **master fields** in **some master record**.
- A Relationship can only be established from **one, and only one, master** Table to **one, and only one, slave** Table.
- A given Table can be involved in **many** Relationships, some as **master** Table and others as **slave** Table.
- Relationships are **directional**. This means that it is **not** the same to define a Relationship from Table A (master) to Table B (slave) than to define a Relationship from Table B (master) to Table A (slave).
- The **values of the slave fields** in the **slave records** are a **subset** of the values of the **master fields** in the **master records**.
- It is **very** different to establish **one** Relationship over **two** (or more) field pairs, than establishing **two** (or more) **different** Relationships each of them over **one** field pair.
- It is possible (although very unusual) to establish a Relationship between fields of the same Table. In you do this, the master Table and the slave Table is the same Table.

I finish highlighting that **Relationships** are a **fundamental** concept of (relational) databases. Relationships make the **huge difference** between having a collection of

**unrelated data Tables**, that you could solve with **several spreadsheets**, and a **database** where the **Table records are related among themselves** and **jointly** create a **single and coherent conceptual model of reality**. In saying this, I am assuming that **referential integrity** is required in **all** the Relationships, because this is the way to get the advantages of Relationships.

If you want to know more about Relationships, you may click:

- “C.11.1 What is a Relationship with referential integrity?”
- “C.11.2 What are “one-to-many” and “one-to-one” Relationships?”
- “C.11.3 What is an indeterminate Relationship?”
- “C.11.4 What is a many-to-many Relationship?”
- “C.11.5 Why should I configure Relationships?”

If you want to configure Relationships in MS-Access, you may click:

- “D.9 How do I create and configure my Table Relationships?”.

### **C.11.1 What is a Relationship with referential integrity?**

A Relationship with **referential integrity** is the one in which the database system (e.g., MS-Access) **guarantees** that the slave field value requirements of the Relationship **are always satisfied, including**, after data **updates** of the **master** fields.

**Referential integrity** is therefore the **automatic guarantee** that the slave field value requirements of the Relationship **are always satisfied, including**, after data updates of the **master** fields.

You can only have Referential integrity if the following **two conditions** hold:

- The two fields of **each and every** master-slave field **pair must** have the **same “Field Type” and “Field Size”** properties. The reason for this is pretty clear: if the two related fields **are not** of the same field type and size, it **cannot be guaranteed** that both fields in the pair **store** the same values.
- The **master field(s)** of the Relationship **must have** an associated **index without duplicate values and without Nulls**. It is not enough if a **subset** or a **superset** of the **master** field(s) have an associated **index without duplicate values and without Nulls**: the index **must** be associated to **exactly** the master field(s). The master Table may additionally have **other indexes**.

In case these two requirements are not fulfilled, when you try to create a Relationship with referential integrity MS-Access will **not** do it, and it will show an error message indicating what the problem is.

Let me try to explain the need for the second requirement above by reusing the same

two Tables from “C.11 *What is a Relationship?*”:

T_Subsidiary_Sites		T_Capital_Cities		
City	Country	Capital	State_Province	Country
Beijing	China	Beijing	No_data	China
Bilbao	Spain	Brasilia	No_data	Brazil
Brasilia	Brazil	Buenos Aires	No_data	Argentina
Buenos Aires	Argentina	Madrid	Madrid	Spain
Sidney	Australia	Washington	District of Columbia	United States
Madrid	Spain			
Sao Paulo	Brazil			
Stuttgart	Germany			
Washington	United States			

The arrows represent the **links** that MS-Access creates between **each master record** and **its slave record(s)** to enforce referential integrity. If you now change the value of “Stuttgart” or remove that record in the master Table “T\_Subsidiary\_Sites”, MS-Access will allow you, because it does not have any slave records, and therefore doing that does not violate referential integrity. However, if you change the value of “Buenos Aires” to “Cordoba” in the master Table, MS-Access will either change the value of “Buenos Aires” to “Cordoba” in **all its slave records**, or else, would not allow you to modify the value, showing a message saying that it would violate referential integrity. You can configure which of both actions MS-Access will take when you create the Relationship.

If you insert a new slave record, MS-Access will only allow you to do it if the values in its slave field “Capital” match the value of one of the existing master records. If there is no match, MS-Access will show you an error indicating that the field “Capital” must take a value from the field “City” in an existing record in “T\_Subsidiary\_Sites”. If there is a match, MS-Access will create a link between the newly inserted slave record, and its corresponding master record, in order to enforce referential integrity on any future data update.

A characteristic of Relationships with referential integrity is that its **related field pairs cannot** contain a **Null**. The master fields **cannot** contain **Null** because they **must** have an index **without duplicate values and without Nulls**. Since the **values** of the **slave** fields **must** be **the same** as the ones of the **master** fields in one **master** record, they also **cannot** contain **Null**. Even if the slave fields have not been configured as “**Required=Yes**”, they **cannot** contain **Null**.

There are **two types** of Relationships that **can have** referential integrity: the **one-to-many** Relationship and the **one-to-one** Relationship. The Relationships that **cannot have** referential integrity are called **indeterminate** Relationships. I explain these three types of Relationships in “C.11.2 *What are “one-to-many” and “one-to-one” Relationships?*” and “C.11.3 *What is an indeterminate Relationship?*”.

I finish this section highlighting (again) that **Relationships** are a **fundamental** concept of (relational) databases. Relationships make the **huge difference** between having a collection of **unrelated data tables**, that you could solve with **several spreadsheets**,



and a **database** where the **Table records are related among themselves** and **jointly** create a **single and coherent conceptual model of reality**. My advice is that you configure referential integrity in all the Relationships that allow for it.

### C.11.2 What are “one-to-many” and “one-to-one” Relationships?

**One-to-many and one-to-one Relationships** are the ones where both fields in **each and every master-slave field pair** have the **same data type**, and also, the **master field(s)** of the Relationship have an **index without duplicate values and without Nulls**. The master Table may additionally have **other indexes**.

Notice that this is **exactly the same** definition of a Relationship that **can have** referential integrity that I stated in “*C.11.1 What is a Relationship with referential integrity?*”. Therefore, **one-to-many** and **one-to-one** are the Relationships where you **can have referential integrity**, and vice versa.

A **one-to-many Relationship** is the one where:

- Both fields in **each and every master-slave field pair** of related fields is of the same data type
- The **master field(s)** have an **index without duplicate values and without Nulls**, and also.
- Neither the **slave field(s)**, nor **any subset** of them, **have an index without duplicate values and without Nulls**.

A **one-to-one Relationship** is the one where:

- Both fields in **each and every master-slave field pair** of related fields is of the same data type
- The **master field(s)** have an **index without duplicate values and without Nulls**, and also.
- Either the **slave field(s)**, or **any subset** of them, **have an index without duplicate values and without Nulls**.

The **master** and/or **slave** Tables may **additionally** have **other indexes**, as long as the requirements above are satisfied.

Since the first condition in both cases is the same, I will present this in hierarchical way:

- **Relationships that can have referential integrity: One-to-many and One-to-one**  
Both fields in **each and every pair** of master-slave fields are of the same data type, and also, the **master field(s)** have an **index without duplicate values and without Nulls**.
  - **One-to-many Relationship**  
Neither the **slave field(s)**, nor **any subset** of them, **have an index without duplicate values and without Nulls**.
  - **One-to-one Relationship**  
Either the **slave field(s)**, or **any subset** of them, **have an index without duplicate values and without Nulls**.
- **Indeterminate Relationship (cannot have referential integrity)**  
Either one (or more) **master-slave field pairs** have fields with a **different** data type, **or**, the **master field(s)** do not have an **index without duplicate values and without Nulls**.

The following table presents in a matrixial way the same information that I have just explained:

<b><u>Conditions to determine the type of Relationship that you establish</u></b>		Every <b>pair</b> of related fields has the same <b>data type</b> , and, the <b>master</b> field(s) <b>have</b> an index <b>without duplicate values and without Nulls</b> ?	
		<b>Yes</b>	<b>No</b>
The <b>slave</b> field(s), or any <b>subset</b> of them, <b>have</b> an index <b>without duplicate values and without Nulls</b> ?	<b>Yes</b>	<b><u>One-to-one</u> Relationship (can have referential integrity)</b>	<b><u>Indeterminate</u> Relationship (cannot have referential integrity)</b>
	<b>No</b>	<b><u>One-to-many</u> Relationship (can have referential integrity)</b>	

Remind that the **master** and/or **slave** Tables may **additionally** have **other indexes**, as long as the requirements above are satisfied.

As **any other Relationship**, **one-to-many** and **one-to-one** have the following characteristics:

- The **values** of the **slave** fields in **every** slave record **must** be the same as the **values** of the **master** fields in **some** master record.
- A Relationship can only be established from **one, and only one, master** Table to **one, and only one, slave** Table.
- A given Table can be involved in **many** Relationships, some as **master** Table and others as **slave** Table.
- Relationships are **directional**. This means that it is **not** the same to define a Relationship from Table A (master) to Table B (slave) than to define a Relationship from Table B (master) to Table A (slave).
- The **values** of the **slave** fields in the **slave** records are a **subset** of the values of the **master** fields in the **master** records.
- It is **very** different to establish **one** Relationship over **two** (or more) field pairs, than establishing **two** (or more) **different** Relationships each of them over **one** field pair.
- It is possible (although very unusual) to establish a Relationship between fields of the same Table. In you do this, the master Table and the slave Table is the same Table.

In addition to these characteristics of any Relationship, **one-to-many** and **one-to-one** Relationships share the following **additional common** characteristics:

- They **can have** referential integrity.
- The **master** Table **does not** have **duplicate records**.

- **Every slave** record from the **slave** Table **must be related** to **one, and only one,** **master** record from the **master** Table.
- The **master** Table **may** have **whatever number** of **master** records that **are not related to any slave record** from the **slave** Table.

A difference between a **one-to-many** and a **one-to-one Relationship** is:

- In a **one-to-many** Relationship **each and every record** from the **slave** Table must be related to **one, and only one,** record from the **master** Table.
- In a **one-to-one** Relationship **each and every record** from the **slave** Table must be related to **one, and only one, different** record from the **master** Table.

Expressing this difference in other words:

- In a **one-to-many** Relationship **each master** record can have **zero or more slave records**, but, in a **one-to-one** Relationship, it can have **zero or one slave** record.

Most likely you think that all of the above seems like a tongue twister, so I better show this with an example.

I will build the example over the Tables “T\_Capital\_Cities” and “T\_Subsiary\_Sites” from *C.11*, plus the Table “T\_Capital\_Rainfall\_Q” from *D.6.5*. I am copying here the Table “T\_Capital\_Rainfall\_Q” for your convenience:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

Let us first think of a Relationship from **master** Table “T\_Subsiary\_Sites” to **slave** Table “T\_Capital\_Cities”. Suppose “T\_Subsiary\_Sites” lists **all** the cities where your company has a subsidiary, and because of this, these cities are **the only cities of interest** in your whole database. Then, it makes all the sense that the **slave field** “Capital” from **slave** Table “T\_Capital\_Cities” take its values from the **master field** “City” in a record from the **master** Table “T\_Subsiary\_Sites”. By “linking” both fields you avoid typing errors (e.g., mistyping the name of a city when inputting it in the **slave** Table) and reference errors (e.g., having a Capital city where you do not have any subsidiary). Since both “City” and “Capital” are a **candidate Key** (click *C.10.3*) in their respective Tables, this is a **one-to-one** Relationship. Notice you may have Cities where you have a subsidiary that **is not** a capital, but, **every** “Capital” in the Table “T\_Capital\_Cities” **must be** in the Table “T\_Subsiary\_Sites”. At the same time, each **master** record from “T\_Subsiary\_Sites” can have either **zero or one slave** records in the Table “T\_Capital\_Cities”. This is exactly what you want, and what you get with this **one-to-**

**one** Relationship.

Let us now think of **another** Relationship, in this case from **master** Table “T\_Capital\_Cities” and **slave** Table “T\_Capital\_Rainfall\_Q”. From the previous example you know that “T\_Capital\_Cities” lists **all** the **Capital** cities where your company has a subsidiary. Then, it makes all the sense that each Capital city in the Table “T\_Capital\_Rainfall\_Q” **must** be listed in the Table “T\_Capital\_Cities”. To achieve this, you just create a Relationship from the **master** field “Capital” from “T\_Capital\_Cities” to the **slave** field “Capital” from “T\_Capital\_Rainfall\_Q”. Since “Capital” **is a candidate Key** in “T\_Capital\_Cities” and “Capital” **is not a candidate Key** in “T\_Capital\_Rainfall\_Q”, this is a **one-to-many** Relationship. Notice each **master** record from “T\_Capital\_Cities” can have **any number** of **slave** records in the Table “T\_Capital\_Rainfall\_Q”. This is exactly what you want, and what you get with this **one-to-many** Relationship.

I finish this section highlighting (once more!) that **Relationships** are a **fundamental** concept of (relational) databases. Relationships make the **huge difference** between having a collection of **unrelated data tables**, that you could solve with **several spreadsheets**, and a **database** where the **Table records are related among themselves** and **jointly** create a **single and coherent conceptual model of reality**. I am assuming that referential integrity is required in all the Relationships.

### **C.11.3 What is an indeterminate Relationship?**

An **indeterminate** Relationship is the one where either one (or more) **pairs** of related fields is **not** of the same data type, **or**, the **master field(s) are not a candidate Key** of the **master** Table. In other words, an indeterminate Relationship **cannot** have referential integrity.

In “C.11.2 *What are “one-to-many” and “one-to-one” Relationships?*” I showed the conditions for the three types of **bilateral** Table Relationships, so you can check also the differences in respect to **one-to-many** and **one-to-one** Relationships.

MS-Access allows you to establish indeterminate Relationships, but you **cannot request referential integrity on them**.

Since indeterminate Relationships cannot guarantee referential integrity, their value is extremely limited. I advise you do not use indeterminate Relationships.

### **C.11.4 What is a many-to-many Relationship?**

A **many-to-many** “*Relationship*” consists of **two, or more, one-to-many** Relationships established **each of them** from one **master** Table to one **common slave Table**. The **common slave** Table is called the **junction** Table of the **many-to-many** “*Relationship*”. As you may see, a **many-to-many** “*Relationship*” **is not** an actual Relationship (it does not comply with the definition of Relationship that I gave). However, it is called a “*Relationship*” because conceptually performs a similar function to an actual Relationship. A **many-to-many** “*Relationship*” among “**n**” Tables is therefore a **set of “n” one-to-many Relationships** each of them between **each of the “n” Tables** and the **junction Table**.

In a **many-to-many** “*Relationship*”, **each** record in **each** of the **master** Tables may be linked with **many** records (even all of them!) from **each of the other master** Tables.

Establishing a **many-to-many** “*Relationship*” involves **more than two Tables**, and as I have already explained, it is not a “proper” Relationship. For these reasons, the **many-to-many** “*Relationship*” is **not** shown as one of the Relationship types in the Relationship boxes of MS-Access, although you can create them as I will explain now.

To create one **many-to-many** “*Relationship*” between “**n**” **master** Tables, you create **one** specific common **slave** Table (the **junction** Table) and you then create a **one-to-many** Relationship between **each** of the “**n**” **master** Tables and **this junction** Table. As you can see, the **junction** Table is the **common slave Table** to **all** of the **master** Tables in the **many-to-many** “*Relationship*”.

When you create the **junction** Table for a given **many-to-many** “*Relationship*”, you have to include in the **junction** Table “**n**” fields (**one field** for **each master field**). Then, you create “**n**” **one-to-many Relationships**, one from **each master** Table to the **junction** Table **over the corresponding field pairs**.

In the **junction** Table, you **may** also include some **additional** fields on top of the “**n**” **mandatory slave** fields. These **additional** fields can provide some supplementary information you may need for each record. Each **junction** Table usually has a **Key** composed of **all its slave fields**, because this is what should be unique in it.

Let me clarify all the above with an example. Imagine you have a Table “T\_Umbrella\_Models” that lists the models of umbrellas that your company manufactures. This Table could look like:

T_Umbrella_Models
Umbrella_Model
Luxurius
Transparent
Ultra Light
Very Large
Wind Resistant

Now, you would like to register in your database **what models of umbrellas you are selling in each Capital city**. It is difficult to do this with a Table, because the number of Capital cities and the number of umbrella models is variable along time and you cannot have Tables with a variable number of fields. Also, the number of capital cities may be very large, and it would be cumbersome to have a Table with so many fields. The solution for this case is a **many-to-many** “*Relationship*”. You create a **junction** Table called “T\_Umbrellas\_in\_Capitals”. This **junction** Table has the fields “Capital” and “Umbrella\_Model”. The **Key** of this Table is its two fields. You now create a **one-to-many** Relationship from **master** field “Capital” of “T\_Capital\_Cities” to **slave** field “Capital” of “T\_Umbrellas\_in\_Capitals” and **another one-to-many** Relationship from **master** field “Umbrella\_Model” from “T\_Umbrella\_Models” to **slave** field “Umbrella\_Model” in “T\_Umbrellas\_in\_Capitals”. You have therefore created a **many-to-many** “*Relationship*” between Tables “T\_Capital\_Cities” and “T\_Umbrella\_Models”. Notice you can now introduce in this Table every possible combinations of umbrella models sold in each Capital city. One example of the values

of “T\_Umbrellas\_in\_Capitals” could be:

T_Umbrellas_in_Capitals	
Capital	Umbrella_model
Beijing	Luxurius
Brasilia	Transparent
Brasilia	Very Large
Madrid	Transparent
Madrid	Ultra Light
Washington	Luxurius
Washington	Very Large
Washington	Wind resistant

I want to highlight the following characteristics of **many-to-many** “*Relationships*” that are **different** from the ones of actual Relationships:

- **Many-to-many** “*Relationships*” are not actual Relationships, but they are called Relationships because conceptually they perform a similar function as an actual Relationship.
- **Many-to-many** “*Relationships*” are **not directional**, in the sense that establishing a “*Relationship*” from Table A to table B is **the same** as establishing it from Table B to Table A.
- **Many-to-many** “*Relationships*” are **not** bilateral. Therefore, you may establish **one many-to-many** “*Relationship*” between **more than two Tables**.
- The Tables related with a **many-to-many** “*Relationship*” may have a **different number of related fields**, and also, the **related fields** can be of **completely different data types**.

### C.11.5 Why should I configure Relationships?

You may feel that Relationships are an unnecessary complication, but this is absolutely **not true!** Relationships are something you configure only once, when you design your database Tables, and out of that limited effort you get **great benefits** in terms of:

- Guarantees of data integrity and coherence.
- Automatic updates of modified master fields.
- Avoid errors arising from mistyped information.
- Faster processing of Queries.

Relationships are **one of the main advantages** of using a database instead of a using a sparse collection of spreadsheets to store and exploit your valuable data. Relationships may be a little tricky to understand at first, but you will quickly grasp the concept and then you will see their meaning and implications crystal clear. Once you have got the concept, Relationships are quite easy to configure in MS-Access.

You should therefore configure as many Relationships as it makes sense among your databases Tables.

## PART D. DESIGNING MY DATABASES WITH MS-ACCESS

The **lifecycle** of a database consists of the following **three phases** of which phases **b)** and **c)** are usually repeated many times:

- a) **Design** your database  
Click “*How do I design my database?*” (see below).
- b) **Use** your database  
Click “*How do I use my database?*” (see below).
- c) **Evolve** your database design  
Click “*How do I evolve my database design?*” (see below).

### How do I design my database?

Designing your database implies the following **steps**. I indicate for each **step** the section where you can find guidance about it.

#### Create the database file and naming policy:

1. **Create** your database **file**  
Click “*D.1 How do I create, close and open a database file?*”.
2. Design a **good naming policy** for the database objects/properties  
Click “*D.2 How do I carefully assign good names from the very beginning?*”.

#### Create all the database Tables:

3. Create each **Table** and its **fields**  
Click “*D.3 How do I create and design a Table and its fields?*”.
4. Configure **each field’s data type**  
Click “*D.4 How do I configure a Table field data type and size?*”.
5. Configure **each field’s properties**, including **simple indexes**  
Click “*D.5 How do I configure a Table field validation rule, indexing, and other properties?*”.
6. Configure the Table **Primary Key field(s)**  
Click “*D.6 How do I configure the Primary Key field(s) of a Table?*”.
7. If you need a **composite index(es)** in the Table, configure it/them  
Click “*D.7 How do I add simple and/or composite index(es) to a Table?*”.
8. Configure the Table **properties** (record validation rule, ...)  
Click “*D.8 How do I configure the properties of a Table?*”.

#### Configure the Relationships and other database aspects:

9. Configure **Relationships** between the Tables  
Click “*D.9 How do I create and configure my Table Relationships?*”.
10. Usually adding **Forms**  
Click “*D.10 How do I design MS-Access Forms?*”.
11. Usually configuring **drop-down menus**  
Click “*D.11 How do I configure the way to enter data (e.g., a drop-down menu)*”.

*in a Table/Form field?”.*

## 12. Possibly adding Reports

Click “*D.12 How do I use MS-Access Reports?”*.”.

## 13. Usually configuring concurrent access

Click “*D.13 How do I share a database, having multiple concurrent users?”*.”.

You should devote **substantial time** to your **initial database design** because **each** database **design change** takes considerable **effort** and creates a **risk** of introducing **errors** in the database data. The reason for this is that the database elements are **very interrelated**. Even a seemingly minor change in a database element (e.g., changing a field name in a Table) usually implies **many modifications** (e.g., adjust accordingly **all the Queries** that use that Table, changing Relationships, ...). You may click “*I.2 Why should I be so careful with any change to the database design?”*.”.

Even if you did a perfect database design when you created the database, you will be modifying its design **for sure**. This may happen (for example) because you (or your customer) want to add more functionality to the database, or because the environment/regulation has changed (accounting, VAT, labor, ...) and this implies changes in how the database works.

Therefore, you should carefully design your database and invest substantial effort in its initial design. You should design to **avoid unnecessary** future **changes** (e.g., by doing a well thought initial design) and to **facilitate** future **changes** that will be **needed** (e.g., by doing an organized and structural design, adding comments, ...).

A good way to do your initial database design is doing it incrementally. You start creating some Tables and Relationships. You paste/input some test data to check if everything seems right. You then do some modifications to the Tables and Relationships, and you add a couple more Tables. You paste/input some more test data to check if everything seems right. You repeat this cycle a few times, until you consider that the database design is sufficiently sound. You then remove all data used for testing and input the correct initial data. The database is then operational, and you begin using it.

## **How do I use my database?**

Using a database consists of **inputting records** into the database Tables, **coding Queries** to exploit the data in the database, and **running the Queries** to obtain the desired summaries, statistics, reports, etc. out of your database.

When you first create the database, you usually upload to it some amount of current and past information records. This usually implies reformatting existing information to adapt it to the design of the database and pasting it (click “*E.5.2 How do I paste data into MS-Access?”*”) in blocks to the different database Tables. When doing this you have to be careful with the consistency of the past information records that you are uploading. If you want to know more about this, you may click “*E.9 Can I get inconsistent results out of my initial data in my database?”*.”.

You then code some initial Queries to obtain relevant reports out of the database. One Query may provide total expenses by category per year, another one the listing of employees in any given year, or whatever complex information you may want to get.

The lifecycle of the database continues as you input records arising from day-to-day



activity, and you further create/modify/delete Queries as to provide the information you want to get from the database.

You may find advice on entering information into your database by clicking:

- “*Part E. Entering, modifying and deleting my database*”

You may find advice on writing SQL Queries by clicking:

- “*Part F. Writing SQL Queries to use my database*”

## **How do I evolve my database design?**

In theory, you do a perfect initial database design, and you never modify it. This **never** happens. You will **for sure** modify your database design along time. Modifying the database design may require a considerable amount of work because the database elements are **very interrelated**. This means that a change in a database element (e.g., changing a field name in a Table) may imply **many modifications** (e.g., adjust accordingly **all the Queries** that use that Table).

You may find advice on how **carefully** evolve your database design by clicking:

- “*Part I. Evolving my database design*”

## **D.1 How do I create, close and open a database file?**




You may click:



- “*D.1.1 How do I create a database file?”*”
- “*D.1.2 How do I close a database file?”*”
- “*D.1.3 How do I open an existing database file?”*”



### **D.1.1 How do I create a database file?**

Before creating your first database file I advise you to set some MS-Access options: click to read “*B.1 What options should I set in MS-Access?”* and then return here (you return by simultaneously pressing the “**Alt**” and “**←**” keys).

To create a database file, open MS-Access, and then you have the following ways:

- If you see the “**New blank database**” button (top left)  
You click on it, and you get a file selector. Select the database file **location** by clicking on the corresponding directories of the file selector. The default file name is “Database1.accdb” or something similar. Click on the default file name to select it and edit the file name to the one you want. It is essential that you **do not** change the file extension (i.e., the suffix of the file name after the period), that **must** remain to be “.accdb”.
- If you see the “ New” button (top left)  
You have the following two options:
  - Click on “” and this will show the “**New blank database**” button: you now follow the instructions in the **first bullet** above.
  - You click on the **Blank Database** “” **large** icon and you get the “**Blank database**” sub-window: you now follow the instructions in the **next paragraph**.


- You click on the **New**  button (top left) and you get the **Blank Database**  large icon. You click on it, and you get the **“Blank database”** sub-window: you now continue in the last bullet.

If you got the **“Blank database”** sub-window (see bullets 2 and 3 above), click on the  icon and you get a file selector. Select the folder for your database file by clicking on the corresponding directories of the file selector. When you are done, click on **“OK”**. The default file name is shown in a box below the text **“File name”**: it should be **“Database1.acdb”** or something similar. Click on the default file name to select it and edit the file name to the one you want. It is essential that you do not change the file extension (i.e., the suffix of the file name after the period), that must remain to be **“.acdb”**. When you are done you click on the **Create**  button, and your new database file has been created.

Regardless of the path you followed among the ones described above, you should now have an open database file.

Notice that when you have an open database file, you **will also see** in the same folder an **additional** file with the **same file name** but with extension **“.laccdb”**. This is an **auxiliary lock** file that MS-Access creates to signal that the database file is open. This auxiliary lock file will **disappear** when you **close** your database file.

### **D.1.2 How do I close a database file?**

To **close** the current database **file**, and also, **close MS-Access**, click on the **close**  icon on the rightmost side of the top frame (the one colored in dark red) of the MS-Access window. This will close the file **and**, will also close the MS-Access window.

To **close** the current database file, **but**, keeping the MS-Access window open, click on **“File”** (at the top left) and then click on **“Close”**. This will close the file, but MS-Access will remain open.

When you **close** the file (with either option above), in case there were **any non-saved objects**, MS-Access will **ask** you if you want to save them **before** closing the file.

### **D.1.3 How do I open an existing database file?**

You can **open** a database file in either of the following ways:

- Double-click on the database file in the Windows File Explorer.
- Open MS-Access like any other windows program. MS-Access will show at the left part of its window a **list** of recently used database files. If you click on any of them, it will be opened. If none of them is the one you want, click on the folder icon at the bottom-left corner (labeled **“Open Other Files”**). This will open the **“Open File”** view of MS-Access.

In the **“Open File”** view, on the right side, you have a list of recently used database files. If you click on any of them, it will be opened. If none of them is the one you want, click on the folder icon (labeled **“Browse”**) at the top-left corner. This will open the usual file-selector window from Windows, where you can navigate the folders until you find the file you want. Once you click on the file you want, the file will be opened in the MS-Access window.

- If you **have** an opened MS-Access **file**, you can **close** it and **open** another file by doing the following. Click on “**File**” on the top “**Ribbon-bar**”, and then click on “**Open**”. This will open the “**Open File**”-view of MS-Access. You now follow the procedure I have just described in the second paragraph of the previous bullet point.

Regardless of what way you follow, you will have an MS-Access window with the database file you wanted opened in it.

When you have an open database file, you will **also** see in the same folder an **additional** file with the **same** file **name** but with the **different extension** “.**laccdb**”. This is an **auxiliary lock file** that MS-Access **creates** to signal that the database file is open. This auxiliary lock file will **disappear** when you **close** your database file.

## **D.2 How do I carefully assign good names from the very beginning?**

You should **carefully think** the **names** of your objects/properties from the **very beginning**. Notice that changing later the name of a field, Table, Query, SQL operation or function will require **substantial adjustment work** over different objects across your database. If you do not assign good initial object names, **do not believe you will easily change them later**: better think them carefully from the very beginning. If you want to know more about the work arising from changing an object name, you may click “*Part I. Evolving my database design*”.

Assigning your object names based on **well-thought, clear** and **regular naming criterion** will save you **a lot of time** of database maintenance and error fixes. It **definitively** pays off to devote some time to decide initially your object names.

My advice on how to choose good names is the following:

- For **all** names, use **only English letters** (“**a**” to “**z**” and “**A**” to “**Z**”), **digits** (“**0**” to “**9**”) and **underscores** (“**\_**”), (click *D.2.4*).
- For **all** names, **begin** them with an **English letter** (click *D.2.4.1*).
- For **all** names, make them **sufficiently different** from **keywords** (click *D.2.4.5*).
- For **Tables** and **Table links**, **begin all** names with “**T\_**” and avoid the suffix “**\_n**” where “**n**” is an integer (click *D.2.4.6*).
- For **Queries** and **Tables**, use **hierarchically structured** names (click *D.2.3*).
- For **fields**, use names that are **short** but **clear** (click *D.2.1*), and **homogeneous** across all your Tables and Queries (click *D.2.2*).
- **Avoid** assigning the **same** name to **different** objects/properties (click *D.2.6*). Notice that for field names the policy is not this one, and you should rather use homogeneous names (see the previous bullet point).

Some advantages of following this naming practice are:

- **Avoid** having to **enclose** your names in **square brackets** “[ ]” **each and every time** you use them. For example, if you use hyphens “-” in names, you **must** enclose such names in square brackets “[ ]” each time you use them, to avoid the ambiguity with the minus “-” operator.

- **Prevent** compatibility **errors** when cutting/pasting or importing/exporting from/to other applications.
- **Easily** distinguishing a Table from a Query along all your SQL code. This distinction is important because Tables typically require using the most restrictive “**WHERE**” expressions (click *K.7.3.1*), and they terminate the object dependency hierarchy, when analyzing side effects (click *I.2.2*).
- **Better maintainability** of your database and SQL Query code.

You should be aware that MS-Access **ignores the case** of the letters in names (this is called “case insensitive”). This means that the following names “US\_Cities”, “us\_cities”, “US\_cities”, “us\_cities” and “US\_CITIES” are **all** considered the **same name** by MS-Access. Also be aware that MS-Access **operators** and **functions** that compare text strings also **ignore the case**<sup>20</sup> in their comparisons.

You may click:

- “D.2.1 Why should I make field names short but clear?”
- “D.2.2 Why should I use homogeneous field names across Tables and Queries?”
- “D.2.3 Why should I use structured names for Tables and Queries?”
- “D.2.4 Why should I avoid certain elements in names?”
- “D.2.5 What are the MS-Access formal rules for identifiers?”
- “D.2.6 When can I assign the same name to different objects and/or properties?”

### D.2.1 Why should I make field names short but clear?

Because this will much facilitate using correctly your database, correcting database errors, and doing database updates.

For **field names** in **Tables** and in **Query results**, try to be **as short as possible, but no more**. Notice that you will want to visualize Tables and Query results in “**Datasheet View**”, and long field names make your columns wider, reducing what you can see on the screen. Of course, you can make the columns narrower than the field name, **but** in this case, you will **not** see the complete field name, and therefore it does not make much sense to have the long field name if you cannot see it. An example of too short field name would be: “Inv\_Dt” (for Invoice Date), because it is too difficult to understand (it is not very meaningful). An example of a field name that is too long would be:

“Date\_in\_which\_the\_invoice\_was\_paid”

Recall that Table fields have a “**Description**” property (click *D.5.1.1*) where you should write what is the detailed meaning of each Table field.

For **internal field names within the SQL** code of a Query, you can be more verbose and use field names as long as you consider necessary. Recall that the **maximum** name length is **64** characters. The only drawback of long field names inside Query code is that you have to type-in more characters when writing the Queries, and Query code becomes

---

<sup>20</sup> The VBA editor has an option to make string comparison “case sensitive” in VBA. My advice is you **do not use this option**, in order to have case sensitiveness **coherence** between your SQL scope and your VBA scope.

longer. Clarity is usually more important than saving keystrokes, so I suggest going to names as long as necessary (but not more) to have a clear indication of what each field exactly is.

Another useful tip for writing field names in Tables and Query results is putting first (i.e., leftmost) the most specific part of the field. The reason is that if you narrow a column to gain view capacity in your screen, the first part of the field name is still visible. For example, it is usually better to use field names “Start\_Project”, “End\_Project” than “Project\_Start”, “Project\_End”. The reason is that if you narrow both columns in “**Datasheet View**”, in the first case you see something like “Start\_Pr” and “End\_Pr”, while in the second case you see “Project\_” and “Project\_”, which is obviously worse.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “*D.2 How do I carefully assign good names from the very beginning?*”.

### **D.2.2 Why should I use homogeneous field names across Tables and Queries?**

Because it will be easier to recall when writing Queries, and also, is also less error prone. For example, if you have a field called “Person” that contains a person full name, it is good to also call that field “Person” in all the Tables that will have that same person identifier, instead of using different field names like “People”, “Full\_name” or “Employee”. In particular, if two fields are **linked** as master-slave in a Relationship (click *C.11*), it is **extremely** convenient to **use exactly the same field name** in both Tables.

For example, if you have a field with the person full name in a Table for intern data, use “Person” better than “Intern” for this field name. The same if you have a Table of students, use “Person” better than “Student” as field name. This will make easier and less error prone the task of writing and maintaining the SQL code of Queries. Notice that within the SQL code you can rename the fields, so **within** the Queries you can rename “Person” to “Student” or “Intern”, although my advice is not to do it.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “*D.2 How do I carefully assign good names from the very beginning?*”.

### **D.2.3 Why should I use structured names for Tables and Queries?**

Because this will make your database much easier to use. On the one hand, because it will list related Tables and Queries together in the “**Navigation Pane**” (if you sort them alphabetically). On the other hand, because the prefixes will make it easier for you to remind the Table or Query name you wanted.

I already advised that you prefix **all** Table names with “T\_” (click *D.2*) because this only adds two characters to your Table names and brings the **big advantage** of **clearly** distinguishing a Table from a Query along your SQL code. This distinction is important because Tables typically require using the most restrictive “**WHERE**” expressions (click *K.7.3.1*), and they terminate the object dependency hierarchy, when analyzing side effects (click *I.5.3*).

It is better to prefix Table names with “**T\_**” than prefixing Query names with “**Q\_**” because usually there are far fewer Tables than Queries, and therefore you will need more characters to structure your Query names.

As another advice, use **common prefixes** in the names of **related** Queries/Tables. In this way, the Tables related among themselves and the Queries related among themselves will be listed contiguously (when sorted alphabetically) in the “**Navigation Pane**”. For example, if you have several Tables that contain information about projects, you may prefix all these Table names with “Proj\_”. The same about Queries: if you have several Queries that provide information on Costs, you may prefix all the Query names with “Cost\_”.

You can also apply the principle in my second advice in a **hierarchical** way, with two or more successive prefixes, in order to group in a hierarchical way the Queries related among themselves and the Tables related among themselves. Place the most generic prefixes first (i.e., leftmost) so objects will be ordered hierarchically in the “**Navigation Pane**” (when sorted alphabetically).

As an example of how to assign **structured hierarchical** Query names, the names of Query examples of this Lighting Guide start with the letter of the *Part* where they are used (e.g., “**A\_**”, “**F\_**”, ...), then they have the operation or concept that they are illustrating, and finally they have a more specific aspect of the concept they are illustrating and/or the sequence in which I show them.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “*D.2 How do I carefully assign good names from the very beginning?*”.

#### **D.2.4 Why should I avoid certain elements in names?**

In this section I explain why some naming practices are a bad idea. If you want my advice on good naming practice, you may click “*D.2 How do I carefully assign good names from the very beginning?*”.

You may click:

- “*D.2.4.1 Why should I avoid special characters in names?*”
- “*D.2.4.2 Why should I avoid “<>” as a field name?*”
- “*D.2.4.3 Why should I avoid non-English letters in names?*”
- “*D.2.4.4 Why must I avoid control characters in names?*”
- “*D.2.4.5 Why must I avoid using keywords as names?*”
- “*D.2.4.6 Why should I avoid the suffix “\_n” in Table names?*”

##### **D.2.4.1 Why should I avoid special characters in names?**

Because special characters (other than “\_”) in names obliges to enclose names in square brackets, can cause problems in SQL code and when importing/exporting code or data from/to other applications. Special characters are characters other than letters (“A”, “b”, “á”, “é”, “C”, “ö”, ...) and digits (“0” to “9”). Examples of special characters are space “ ”, hyphen “-”, comma “,”, at-sign “@”, number-sign “#”, ampersand “&” and similar ones.

Notice that this implies that you **should not** use **spaces** in your object names. As an **exception** to avoid using special characters, you **should** use underscore “\_” in your names (but **not** as the first character) as a **replacement** for spaces in your object names is a **good practice**.

Notice that it is **mandatory** to enclose the **name** in square brackets “[ ]” in the following cases:

- When **invoking** (but **not** when **assigning**) **field names** or **SQL operation names** starting with “\_” or with a digit (“0” to “9”).
- When **invoking** or **assigning field names** containing any **special character** other than “\_”.

Finally, the characters period (“.”), exclamation (“!”) and square brackets (“[” and “]”) are completely **forbidden** within object names. These characters are forbidden even if you enclose the object name in quotes and/or in square brackets. You **cannot** use any of these characters in your object names.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 How do I *carefully assign good names from the very beginning?*”.

#### **D.2.4.2 Why should I avoid “<>” as a field name?**

I already advised against using special characters (except “\_”) in field names. For the case of **specifically** avoiding the field name “<>”, the reason is that MS-Access uses it to represent a **Null** as a “**PIVOT**” **field name** in **Transform Queries**. For this reason, I strongly advise you **avoid using** “<>” for **any** of your field names, to avoid possible confusion with “**PIVOT**” **field names** in **Transform Queries**.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 How do I *carefully assign good names from the very beginning?*”.

#### **D.2.4.3 Why should I avoid non-English letters in names?**

Because in spite of MS-Access handling them **perfectly right**, and not requiring to enclose such names in square brackets, using non-English letters in names can possibly cause some problems in SQL code and when importing/exporting from/to other applications. By non-English letters I mean vowels with different types of accents (e.g., “á”, “à”, “ê”, “ö”, “ø”, ...) and non-English consonants (“ç”, “ñ”, ...).

MS-Access considers that all the English and non-English letters as different. Some examples follow. The identifiers “cases” and “çases” are not the same. The identifiers “nues” and “ñues” are not the same. The identifiers “cases”, “câses”, “càses”, “câses” and “cäses” are all different.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 How do I *carefully assign good names from the very beginning?*”.

#### **D.2.4.4 Why must I avoid control characters in names?**

Because control characters are **completely** forbidden in object names. These characters

are forbidden even if you enclose the object name in quotes and/or in square brackets. You **cannot** use any control character in your object names.

Control characters are ASCII decimal values 000 up to and including 031. These are non-printable characters like **escape**, **new-line**, **horizontal tab** and similar ones, used to convey a special meaning on many programs.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 How do I *carefully assign good names from the very beginning?*”.

#### D.2.4.5 Why must I **avoid** using **keywords** as **names**?

Because if the name matches exactly a keyword, it will be rejected by MS-Access, and if it is very similar to a keyword, it will create confusion in your SQL code.

Using an MS-Access keyword (e.g., “From”, “Select”, “Integer”, “Null”, “True”, “In”, “Text”, “Date”, ...) as a **name** will be usually (depending on context) rejected by MS-Access, showing an error message. For example, **all** the following SQL Query **field names** (highlighted in brown color) are not accepted by MS-Access.

```
SELECT #1/1/2010# AS Date FROM ...
SELECT #Text_tex# AS String FROM ...
SELECT 78 AS Integer FROM ...
SELECT 34 AS Join FROM ...
SELECT 34 AS From FROM ...
```

If you try any of the field name assignments above, MS-Access will not allow you to save the Query, showing the error message “*The SELECT statement includes a reserved word or an argument name that is misspelled or missing, or the punctuation is incorrect.*”

You should **also avoid** object names that are **too similar** to a keyword. Using names that are not exactly a keyword, but very similar to a keyword (e.g., “From\_”, “Null\_”) will not be rejected by MS-Access. However, using that type of names is a **bad practice** because it makes your SQL more difficult to read by creating some confusion.

It is not a problem to use a keyword as **part** of a name, as long as the result is sufficiently different from the keyword so as to avoid confusion) For example, “Selected\_city”, “Integer\_part”, “Null\_value” are **valid** and **reasonable** object names.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 How do I *carefully assign good names from the very beginning?*”.

#### D.2.4.6 Why should I **avoid** the **suffix** “\_n” in **Table names**?

Because MS-Access **adds** the suffix “\_1”, “\_2”, “\_3”, etc. to a **Table name** in order to build the **names** of the different **Table-boxes** representing **that Table** in the “**Relationships**” pane (click *B.10.4.4*). If you also use the suffixes “\_1”, “\_2”, etc. for Table names, this creates the potential for confusion in the “**Relationships**” pane.

If you really need to name a series of Tables with sequential suffixes, use “\_a”, “\_b”, “\_c”, etc.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and



“A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 How do I *carefully assign good names from the very beginning?*”.

### D.2.5 What are the MS-Access formal rules for identifiers?

This section presents the formal MS-Access rules for identifiers. On my view, these formal rules are much more complex than what is needed on 99.9% of the cases and can encourage you to use features that are allowed but can create you some usability or compatibility problems.

My advice is therefore that you skip this section and read instead “D.2 How do I *carefully assign good names from the very beginning?*”.

The general **formal structure** of an MS-Access identifier is:

```
{[]Collection_name[]}!{[]Object_name[]}{.[]}Property_name{[]}
```

where curly braces indicate optional parts. “Collection\_name” denotes the name of a given object collection. “Object\_name” denotes the name of a given object. “Property\_name” denotes the name of a specific object property. The name-separation characters “!” and “.” are required **only** if **both** names to its sides are present and should be **omitted** otherwise.

A “**Collection**” is a group of database objects of the same class. Some examples of collections are: Table, Query, Form and Report.

An “**Object**” is each Table, each Query, each SQL operation in your Query code, each Form, each Report and each VBA module. Each time you create an object (with the exception of SQL operations) you **must** assign it a name.

Objects have “**Properties**”, which describe, and provide a way to change, the object's characteristics. For example, each **field** in a Table or Query object is a **property** of the Table or Query.

The MS-Access **rules** for defining each **object/property** name are the following:

- Can be up to **64** characters long.
- Can include any combination of:
  - English letters and “a” to “z” and “A” to “Z”)
  - Digits (“0” to “9”)
  - Foreign letters (e.g., “ñ”, “Ñ”, “ó”, “Ó”, “è”, “È”, “ç”, “ü”, “Ü”, etc.)
  - Special characters (e.g., space “ ”, “@”, “#”, “\$”, “%”, underscore “\_”, hyphen “-”, etc.), **except** the following  
**forbidden ones**: period “.”, exclamation point “!”, accent grave “`”, and square brackets “[” and “]”.
- **Cannot** include **control characters** (ASCII decimal values from 000 up to and including 031).
- **Cannot begin** with leading **spaces**.
- Names are case insensitive. For example, MS-Access will consider “US\_CITIES”, “us\_cities”, “US\_Cities” and “US\_cities” as **the same name**.

You only have to write enough parts (Collection\_name, Object\_name and/or Property\_name) of an identifier to make it unique in the context you are using it.

Actually, the **most frequent** case is that you only need to write the **plain** object or property name, **without** square brackets.

It is **mandatory** to qualify a field name when it is **ambiguous**. Qualifying means prefixing the field name with the object name of the SQL operation (or Table/Query) to which it belongs and a period “.” character (as I have shown above in the general structure for identifiers). For example, the field names “Invoice\_Date”, “Start\_Date” and “End\_Date” are all prefixed in the following expression by their corresponding object names:

```
Invoices.Invoice_Date BETWEEN Projects.Start_Date AND Projects.End_Date
```

An example of such an **ambiguous** situation is the **output expression** of a **Select** operation enclosing a **Join** operation where the **same field name** exists in its two **input** record-lists. This may seem a strange case, but it is actually a **quite frequent case**, and you should qualify the field names in each such **Select** operation.

It is **mandatory** to enclose the **name** in square brackets “[ ]” in the following cases:

- When **invoking** (but **not** when **assigning**) **field names** or **SQL operation names** that start with either “\_” or with a digit (“0” to “9”).
- When **invoking** or **assigning field names** containing any **special character** other than “\_”.

In respect to the rules for name collisions, I present them in the following section “D.2.6 *When can I assign the same name to different objects and/or properties?*”.

My overall advice for **good** names is that you use **only English letters** (“a” to “z” and “A” to “Z”), **digits** (“0” to “9”) and **underscores** (“\_”), plus other guidelines explained in “D.2 *How do I carefully assign good names from the very beginning?*”.

### **D.2.6 When can I assign the same name to different objects and/or properties?**

My advice is you **avoid as much as possible** assigning the same name to different objects and/or properties (except for field names, that should be homogeneous across the database Tables and Queries). However, in case you want/need to do it, the MS-Access rules are the following:

- **Cannot** assign the same name to two Tables in the same database file.  
**Can assign** the same name to two Tables if they are in **different** database files.
- **Cannot** assign the same name to Queries in the same database file.  
**Can assign** the same name to two Queries if they are in **different** database files.
- **Cannot** assign the same name to a Table and a Query in the same database file.  
**Can assign** the same name to a Table and a Query if they are in **different** database files.
- **Cannot** assign the same name to two SQL operations in the same SQL scope.  
**Can assign** the same name to two SQL operations if they are in **different** SQL scopes.
- **Cannot** assign the same name to two fields in the same record-list.  
**Can assign** the same name to two fields if they are in **different** record-list,

regardless of the record-list being a Table, a Query or an SQL operation.

- **Cannot** assign the same name to two parameters in the same “**PARAMETERS**” clause.  
**Can assign** the same name to two parameters if they are in different “**PARAMETERS**” clauses. If you want to know more about parameters, you may click *F.12*.

### D.3 How do I create and design a Table and its fields?

Before creating any Table, I strongly advice to unset the option “**Enable design changes for tables in Datasheet view**” (click *B.1*). This will remove the column “**Click to Add**”. I consider this column very **distracting**. It is also **risky**, because it may cause the user to inadvertently modify the Table structure.

Before creating any Tables, you should **carefully think** about **all** their fields, what should be the **Key field(s)** and also what will be the **Relationships between different Tables** (click “*How do I design my database?*”).

Designing **each** Table implies the following **database design steps** (click *Part D*):

#### 3. Create the **Table** and its **fields**

Click “*D.3 How do I create and design a Table and its fields?*”.

#### 4. Configure each **field’s data type**.

Click “*D.4 How do I configure a Table field data type and size?*”.

#### 5. Configure each **field’s properties**, including **simple indexes**

Click “*D.5 How do I configure a Table field validation rule, indexing, and other properties?*”.

#### 6. Configure the Table **Primary Key field(s)**

Click “*D.6 How do I configure the Primary Key field(s) of a Table?*”.

#### 7. If you need a **composite index(es)** in the Table, configure it/them


Click “*D.7 How do I add simple and/or composite index(es) to a Table?*”.

#### 8. Configure the Table **properties** (record validation rule, ...)

Click “*D.8 How do I configure the properties of a Table?*”.


Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.



### D.3.1 How do I create a Table?

Click “**Create**” from the “**Ribbon-bar**”, and then click on the **Table Design**  icon inside the Ribbon (i.e., the top wide toolbar). This will create a Table and open it in “**Design View**”. When you have the Table opened in “**Design View**” you see at the top of the “**Table pane**” a table in which **each row** represents a **field** of the Table

Once you have created the Table, you should **create** its fields. You may click:

- “*D.3.2 How do I create my Table fields?*”

You can also create a Table by clicking on the **Table**  icon instead of on the **Table**

**Design** “” icon (as indicated above). However, my advice is that you **do not** create a Table using the **Table** “” icon, because this will add a field named “**Id**” (with field type “**AutoNumber**”), and will configure it as the Table’s **Key** field, which almost always is a **bad Key** design.

If you want to know more about the **Key** fields, you may click:

- “*C.10 What are the Table Key(s) and how should I handle them?*”
- “*D.6 How do I configure the Primary Key field(s) of a Table?*”

### **D.3.2 How do I create my Table fields?**

Open the Table in “**Design View**” (click *B.4.1.3*).

**Add fields** to the Table by **typing-in** their **name** in the “**Field Name**” column of the table in the top sub-pane. You **also** have to configure the **data type** of each field by clicking on the rightmost side of the field cell in the “**Data Type**” column. Then click on data type that you want from the drop-down menu. You should also write a description of the field in the “**Description (Optional)**” column. If you want a more detailed description on how to add fields, you may click “*B.6.1.5 How do I add Table fields in “Design View”?*”.

For each Table field that you add, you should also define its field properties (click *D.3, D.4* and *D.5*), whether it is a **Key** field (click *D.6*), and possibly its associated composite index(es) (click *D.7*).

I advise you **always** write field descriptions because it is **essential** to understand **very precisely** the meaning of each field, and it is **not as obvious** as it may seem. For example, imagine you have a contract Table with **Date/Time** fields “**First\_Day**” and “**Last\_Day**”. It is not obvious whether “**First\_Day**” corresponds to the first day of the contract, or the previous day, and the same applies to the field “**Last\_Day**”. As another example, imagine that you have a field called “**Rainfall**”: it is not obvious if its units are inches, pints per triangular foot, millimeters or other.

The order of the fields in the Table for SQL Query purposes is the order of the field rows in Table “**Design View**”. The first field is the topmost one, and the last field is the bottommost one.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

## **D.4 How do I configure a Table field data type and size?**

This chapter also answers the question:

- **What are the available field types in Tables?**

Open the Table in “**Design View**” (click *B.4.1.3*).

Click on any of the rows that represent fields in the top sub-pane. They may be existing fields that you are now editing, or a field you just created. In either case, MS-Access will allow you to select the field type from a drop-down menu shown by clicking on the

rightmost side of the field cell below the heading “**Data Type**”. I now present a list of **field types** that you can configure using this drop-down menu, and for each of them you may click the section where it is explained:

- **Short Text**: “[D.4.1 What is the “Short Text” field type?](#)”
- **Long Text** (formerly “Memo”): “[D.4.2 What is the “Long Text” field type?](#)”
- **Number**: “[D.4.3 What are the “Number” field types?](#)”
- **Currency**: “[D.4.4 What is the “Currency” field type?](#)”
- **Date/Time**: “[D.4.5 What is the “Date/Time” field type?](#)”
- **Date/Time extended** (only after 2020): “[D.4.6 What is the “Date/Time extended” field type?](#)”
- **Yes/No** (similar to *Boolean*): “[D.4.7 What is the “Yes/No” field type?](#)”
- **Calculated** (not a data type): “[D.4.8 What is the “Calculated” field type?](#)”
- **Large Number** (only after 2016): “[D.4.9 What is the “Large Number” field type?](#)”
- **AutoNumber**: “[D.4.10 What is the “AutoNumber” field type?](#)”
- **OLE Object**: “[D.4.11 What is the “OLE Object” field type?](#)”
- **Hyperlink**: “[D.4.12 What is the “Hyperlink” field type?](#)”
- **Attachment**: “[D.4.13 What is the “Attachment” field type?](#)”
- **Lookup Wizard...** (not a data type): “[D.4.14 What is the “Lookup Wizard...”?](#)”

If you **change** the field “**Type**” and/or “**Size**” properties in a Table with existing records, when saving the Table design MS-Access will check the values **existing** in the Table in respect to the new value of the “**Type**” and/or “**Size**” properties (click [I.4.4.1](#)).

Once you are done with your Table configuration, **save** (click [B.4.1.6](#)) your Table design. You may then **close** the Table (click [B.4.1.7](#)) or change it to “**Datasheet View**” (click [B.4.1.4](#)). If you get warning and/or **error** messages when saving your Table design, you may click “[L.2 How do I fix errors with my Table/Form design?](#)”.

### **D.4.1 What is the “Short Text” field type?**

The **Short Text** field type stores a **text string** with a **configurable** maximum length up to a maximum value of 255 characters.

Each **Short Text** field value is stored/represented with a variable number of bytes, depending on the number of characters of the text string.

The **Short Text** field type is **equivalent** to the VBA *String* data type (click [G.2](#)).

I recommend you use the **Short Text** field type for text information you may need to store in the database (person names, company names, product names, comments, concept in invoices, ...).

The default maximum string length is 255 characters. You may modify the maximum string length of each individual **Short Text** field by changing the value of the property “**Field Size**”. You can find the “**Field Size**” property at the bottom of the “Table pane”, in the “**General**” tab. I recommend you keep the default string length of 255 in **all** fields with **Short Text** data type unless you have a **clear and strong** reason to change it. Notice that if you have fields with different lengths, and you produce values from/to these fields, you may find that MS-Access is truncating the length of some values to the one of shorter fields.

Notice that you may select as an MS-Access option the **default** length for **Short Text** fields (click *B.1*).

#### **D.4.2 What is the “Long Text” field type?**

Stores **formatted text** longer than 255 characters.

Each value is stored/represented using a variable number of bytes, depending on the length and other characteristics of the text.).

The **Long Text** field type is **partly** equivalent to the VBA *String* data type (click “*G.2 How do I manage VBA data types and Table field types-sizes?*”).

You **cannot** configure a **drop-down menu** on fields of this type. This field type is called “Memo” in MS-Access versions older than 2013.

#### **D.4.3 What are the “Number” field types?**

The **Number** field type stores **numbers** with different characteristics. The **Number** field type is refined into finer-grain field types to store different types of numbers. You may select the fine-grain data type by changing the value of the property “**Field Size**” that you can find in field properties, placed at the bottom sub-pane of the “Table pane”, in the tab “**General**”. Clicking on the rightmost side of the said “**Field Size**” row will show a drop-down menu with the following options:

- Number-Field Size = **Byte**  
Click “*D.4.3.1 What is the “Number-Byte” field type-size?*”.
- Number-Field Size = **Integer**  
Click “*D.4.3.2 What is the “Number-Integer” field type-size?*”.
- Number-Field Size = **Long Integer**  
Click “*D.4.3.3 What is the “Number-Long Integer” field type-size?*”.
- Number-Field Size= **Single**  
Click “*D.4.3.4 What is the “Number-Single” field type-size?*”.
- Number-Field Size= **Double**  
Click “*D.4.3.5 What is the “Number-Double” field type-size?*”.
- Number-Field Size= **Replication ID**  
Click “*D.4.3.6 What is the “Number-Replication ID” field type-size?*”.
- Number-Field Size= **Decimal**  
Click “*D.4.3.7 What is the “Number-Decimal” field type-size?*”.

In the following subsections I explain each of the cases above:

##### **D.4.3.1 What is the “Number-Byte” field type-size?**

The **Number** field type with **Byte** field size stores **positive integer** numbers,

from: 0  
to: 255

both included. Precision is 3 significant digits.

Each value is stored/represented as an unsigned binary integer using 1 byte (8 bits).

The **Number-Byte** field type-size is equivalent to the VBA *Byte* data type (click *G.2.I*). Unless you really need this for technical reasons, or you really need to save storage space, my advice is **you do not use “Field Size=Byte”** and use instead **“Field Size=Long Integer”**.

#### **D.4.3.2 What is the “Number-Integer” field type-size?**

The **Number** field type with **Integer** field size stores **integer** numbers,

from: -32,768     $-(2^{15})$   
to:    32,767     $(2^{15}-1)$

both included. Precision is 5 significant digits.

Each value is stored/represented as a signed binary integer using 2 bytes (16 bits).

The **Number-Integer** field type-size is equivalent to the VBA *Integer* data type (click *G.2.I*).

Unless you really need to save storage for a very good reason, my advice is **you do not use “Field Size=Integer”** and use **instead “Field Size=Long Integer”**.

#### **D.4.3.3 What is the “Number-Long Integer” field type-size?**

The **Number** field type with **Long Integer** field size stores **integer** numbers:

from: -2,147,483,648     $-(2^{31})$   
to:    2,147,483,647     $(2^{31}-1)$

both included. Precision is 10 significant digits.

Each value is stored/represented as a binary signed integer number using 4 bytes (32 bits).

The **Number-Long Integer** field type is equivalent to the VBA *Long* data type (click *G.2.I*).

My advice is **you use “Field Size=Long Integer”** for all your integer fields.

#### **D.4.3.4 What is the “Number-Single” field type-size?**

The **Number** field type with **Single** field size stores **fractional** numbers in **floating-point** format, approximately,

from:  $\pm 1.17 \times 10^{-38}$   
to:     $\pm 3.40 \times 10^{38}$

Precision is 6 to 9 significant digits.

Each value is stored/represented as a binary signed floating-point number using 2 bytes (16 bits).

The **Number-Single** field type is equivalent to the VBA *Single* data type (click *G.2.I*).

Unless you really need to save storage for a very good reason, my advice is **you do not use this “Field Size=Single”** and use instead **“Field Size=Double”**.

#### **D.4.3.5 What is the “Number-Double” field type-size?**

The **Number** field type with **Double** field size stores **fractional** numbers in **floating-point** format, approximately,

from:  $\pm 9.881 \times 10^{-324}$   
to:  $\pm 1.797 \times 10^{308}$

Precision is 15 to 17 significant digits.

Each value is stored/represented as a binary signed floating-point number using 4 bytes (32 bits).

The **Number-Double** field type-size is equivalent to the VBA *Double* data type (click *G.2.1*).

My advice is you **use “Field Size=Double” for fractional numbers**, and also, for **currency values when** in case you want to have a drop-down menu.

#### **D.4.3.6 What is the “Number-Replication ID” field type-size?**

The **Number** field type with “**Replication ID**” field size stores internal pointers to internal MS-Access objects.

My advice is you **do not use “Field Size=Replication ID”** unless you are a **top expert** in MS-Access.

#### **D.4.3.7 What is the “Number-Decimal” field type-size?**

My advice is you **do not** use “**Field Size=Decimal**” because it may create several problems, errors and Query crashes. If you want to know more about why to avoid **Decimal**, you may click “*K.8 Why should I avoid using Decimal data types?*”.

#### **D.4.4 What is the “Currency” field type?**

The **Currency** field type stores **fractional** numbers in **fixed-point** format,

from: -922,337,203,685,477.5808  $(-(2^{63}) / 10^4)$   
to: 922,337,203,685,477.5807  $((2^{63}-1)/10^4)$

both included. Precision is 19 significant digits, having four decimals.

Each value is stored/represented as a binary signed fixed-point number using 8 bytes (64 bits).

The **Currency** field type is equivalent to the VBA *Currency* data type (click *G.2.1*).

Its **great advantage** is that it does not produce **any** binary/decimal conversion **rounding-errors**. Its disadvantage is that only has four decimals, which makes it unsuitable for fractional numbers. Also, this field type does **not** allow to configure **drop-down menus**.

My advice is, **use this field type only** if you need to avoid binary/decimal conversion rounding errors. Otherwise, use the **Number-Double** field type-size for your fractional numbers.

If you want to know more about rounding errors caused by converting to/from binary/decimal, you may click “*G.9.2 What is decimal/binary conversion?*” and “*G.9.3 What are decimal/binary conversion rounding errors?*”.



### D.4.5 What is the “Date/Time” field type?

The **Date/Time** field type stores a **date-part** and a **time-part**:

- **Date-part** ranges from **1-January-100** to **31-December-9999**.
- **Time-part** ranges from **00:00:00** to **23:59:59**.

Each **Date/Time** value is stored/represented as a binary signed floating-point number using 8 bytes (64 bits).

The **Date/Time** field type is equivalent to the VBA *Date* data type (click *G.2.1*).

This field type **always** stores **both date** and **time**. Even if you enter a **zero-time** value or a **zero-date** value in the field, it will store a **date-part** and a **time-part**.

Although a **Date/Time** field **always** stores a **date-part** and a **time-part**, you can configure its “**Format**” property to **show** only the **date-part**, **show** only the **time-part** or **show both** the **date-part** and the **time-part** of the value. This can **mislead** you to think that it is possible to **store** only date or only time, which is **not** the case.

My advice is, **use this field type** for date and/or time values.

Each **Date/Time** value is stored/represented **internally** by the system as a **fractional number** (in particular, as a **double** precision floating-point **number**). The **integer** part of the fractional number represents the **date** and the **decimal** part of the fractional number represents the **time**. **Each** integer **unit** in the fractional number represents **one day**. Since one unit is one day, equivalent to 24 hours, it follows that the fractional value “1/24” is one hour, decimal value “1/(24\*60)” is one minute and fractional value “1/(24\*60\*60)” is 1 second.

Notice that **Number-Double** values and *Double* values are also stored/represented internally as a double precision floating-point number. This is why **Date/Time** or *Date* values can be **combined** with **Number-Double** or *Double* values in most operators, functions and expressions. If you want to know more about this, you may click “*G.2.1 What VBA data types vs. Table field types-sizes are equivalent?”.*

**Dates** represented within a fractional **number** always advance **forward**: this means that if number *X* is 6 integer units **larger** than number *Y*, it follows that number *X* represents a date that is six days **more** than number *Y*. However, **time** advances **forward** when the fractional number is larger or equal than “**0**”, but it advances **backwards** when the fractional number is less than or equal to “**-1**”. If you want to do **non-integer** arithmetic operations over negative **Date/Time** or *Date* values, make sure you understand **very well** how this works.

Because of the way a **Date/Time** or *Date* value is represented as a fractional number, **there are no Date/Time nor Date values between “-1” and “0”**. Also, adding “.25” to a **Date/Time** or *Date* with numeric value greater than or equal to “**0**” will **always increase** its time in six hours. However, adding “-.25” (i.e., subtracting “.25”) to a numeric **Date/Time** or *Date* value less than or equal to “**-1**” will **increase** its time in six hours **as long as this operation does not change the integer part of the value!!** In case the integer part changes, then the result will **not be** the initial **Date/Time** or *Date* value plus six hours!!

In numeric values, dates range from **-657,434** to **2,958,465**. This is equivalent to the

date range from **1-January-100** to **31-December-9999**. Numeric value **0** is **30-December-1899**.

In numeric values, time ranges from **+/-0** to **+/-999988**. This is equivalent to the time range from **00:00:00** to **23:59:59**.

The numeric value “**0**” is therefore interpreted as the **Date/Time** or *Date* value:

**30-December-1989 0:00:00**

Since the integer part is interpreted as full days, the numeric value “**23**” is 23 days **more** than the value “**0**” above. Therefore, numeric value “**23**” is interpreted as the **Date/Time** or *Date* value:

**21-January-1990 0:00:00**

Likewise, the numeric value “**-38**” is 38 days **less** than the value “**0**” above. Consequently, numeric value “**-38**” is interpreted as the **Date/Time** or *Date* value:

**22-November-1989 0:00:00**

Because numeric value “**1/24**” is one hour, the numeric value “**17.25**”, is interpreted as the **Date/Time** or *Date* value:

**16-January-1990 6:00:00**

Notice in the explanation above that **time** works as an **unsigned** value **both** for **positive** and for **negative** numeric values. Therefore, numeric value “**-2.25**” is interpreted as “**-2**” days **plus** “.25\*24” time. Therefore, numeric value “**-2.25**” is interpreted as the **Date/Time** or *Date* value:

**28-December-1989 6:00:00**

As a final remark, notice that **Yes/No** (and *Boolean*) values and **Date/Time** (and *Date*) values are internally represented as numeric values. For this reason, you can (**carefully**) combine **Yes/No** (and *Boolean*) values, **Date/Time** (and *Date*) values and **numeric** values in your expressions. For example, adding the **integer** number “**1**” to a **Date/Time** value will add one day to the **Date/Time** value.

#### **D.4.6 What is the “Date/Time extended” field type?**

The **Date/Time extended** field type stores a **date-part** and a **time-part**:

- **Date-part** ranges from **1-January-1** to **31-December-9999**.
- **Time-part** ranges from **00:00:00** to **23:59:59.9999999**.

Each **Date/Time** value is stored/represented as an encoded string of 42 bytes (336 bits).

The **Date/Time extended** field type **does not have** any equivalent VBA *Date* data type.

This field type **always** stores **both date** and **time**. Even if you enter a **zero-time** value or a **zero-date** value in the field, it will store a **date-part** and a **time-part**.

The **advantages** of “**Date/Time extended**” over “**Date/Time**” are that it can store the **additional** years from year “**1**” to year “**99**” and that it has time precision down to one nanosecond.

The **disadvantages** of “**Date/Time extended**” over “**Date/Time**” are:

- It **does not** have an equivalent VBA data type.
- It **does not** have an equivalent Excel data type (so you will have difficulties when pasting into Excel).
- It requires **over five times more** storage space (from 8 bytes to 42 bytes).
- It requires **more processing time** when operating over it.

I think the disadvantages are very considerable, so my advice is you avoid using it.

#### **D.4.7 What is the “Yes/No” field type?**

The **Yes/No** field type stores either **True/Yes/On** or **ticked** or **False/No/Off** or **unticked**.

Each **Yes/No** value is stored/represented as a **Number-Integer** field type. Therefore, each **Yes/No** value is stored/represented as a binary signed integer number using 2 bytes (16 bits) of storage space. The value **True/Yes/On** or **ticked** is stored/represented as the numeric value “**-1**”. The value **False/No/Off** or **unticked** is stored/represented as the numeric value “**0**”. You may find it strange to use 2 bytes (16 bits) of storage space to represent only two possible values, but this is the way it works.

The **Yes/No** field type is equivalent to the VBA **Boolean** data type (click *G.2.1*).

My advice is, **use this field type for Boolean/binary information**.

Notice that **Yes/No** values and **Date/Time** values are internally represented as **Number** values. For this reason, you can (**carefully**) combine **all of them** in expressions.

When using a numeric value as a **Yes/No** value (in a field or in an expression), the numeric value “**0**” is interpreted as **False/No/Off** or **unticked**, and **any other** numeric value is interpreted as **True/Yes/On** or **ticked**. Therefore, numeric values “**-1**”, “**-5,000**”, “**-4.567**”, “**1**”, “**37.3455**” are all interpreted as **True/Yes/On** or **ticked**.

If you want to know more about the internal representation of **Number** values you may click “*D.4.3 What are the “Number” field types?*”.

#### **D.4.8 What is the “Calculated” field type?**

This is **not** a field type. This is rather an **expression** that allows to **calculate** the value of this field, typically involving the values of **other fields in the record**. You have to type-in the expression you want to use to calculate this field in the property “**Expression**”, in the field properties placed at the bottom sub-pane of the “Table pane”, in the tab “**General**”.

The **actual field type** of the calculated value is configured in the property “**Result Type**”. Available options for “**Result Type**” are:

- **Double**
- **Integer**
- **Long Integer**
- **Single**
- **Replication ID**

- **Decimal**
- **Short Text**
- **Date/Time**
- **Long Text**
- **Currency**
- **Currency**: MS-Access 365 shows **Currency** twice and does not show **Large Number**. I have reported this as a likely MS-Access **bug**.
- **Yes/No**

The field type of the calculated field will be the one corresponding to the value of the “**Result Type**” property, that you can configure among the ones in the list above.

The **value** of a calculated field is **not stored** in the record: it is **calculated**, from the other field values, **each and every time** you use the record. Using this type of fields has some subtle disadvantages. Before you decide to use a **Calculated** field, I recommend you click “[K.2.6 What is the difference between a Calculated field and automatically introducing a value using a Form?](#)” where I explain calculated fields vs. using Forms.

If you are familiar with the expression syntax in Table “**Design View**” (recall that MS-Access has **three** different expression syntax, click [G.1](#)), you may directly type-in the expression in the property “**Expression**”, in the field properties, placed at the bottom of the “Table pane”, in the tab “**General**”. If you are not familiar with the syntax of these expressions, you click on the icon with three dots “**...**” at the right of this row, and MS-Access will open a dialogue-box to help you write the expression you want.

In the expression of a **Calculated** field you can use constants, built-in operators, built-in VBA functions and all this Table’s field names. However, you **cannot** use user-defined VBA functions, nor Subqueries. If you want to know more about the rules to write expressions in Table “**Design View**”, you may click “[G.1 What are the main differences between the three expression scopes?](#)”.

#### **D.4.9 What is the “Large Number” field type?**

The **Large Number** field type stores **integer** numbers,

from: -9,223,372,036,854,775,808 ( $-2^{63}$ )

to: 9,223,372,036,854,775,807 ( $2^{63}-1$ )

both included. Precision is 19 significant digits.



Each **Large Number** value is stored/represented as a binary integer number using 8 bytes (64 bits).

The **Large Number** field type is **partly** equivalent to the VBA **LongLong** data type (click [G.2.2](#)).

This field type is not compatible with MS-Access 2016 and earlier versions, and may create problems when exporting/importing data, and with other operations (e.g., VBA data types on 32bit Office versions). My advice is **you do not use “Field Type=Large Number”** unless you **really need** to manage integer numbers larger than can be stored by **Long Integer** field size of the **Number** field type.

#### D.4.10 What is the “AutoNumber” field type?

The **AutoNumber** field type stores a **Long Integer** value, with the **very special** characteristic that MS-Access **automatically** assigns a different value to each record you insert in the Table. It therefore **guarantees** no record duplicates.

It may **seem** useful to add to your Table one **AutoNumber** field and configure it as the Table’s **Key** field. Actually, this is what MS-Access does if you create a Table clicking on the **Table** “” icon instead of on the **Table Design** “” icon (click *D.3.1*). However, using one **AutoNumber** field as the **Key** field removes all the semantics from the Table’s **Key** field and is a poor design. Therefore, my advice is that you **do not use** the **AutoNumber field type** unless **you really want** the system to assign different values for you (e.g., when creating members entries in a club, to assign the member number), and you only use it as the **Key** field when there is no other Key with better semantics for it.

If you want to know more about the **Key** fields, you may click:


- “*C.10 What are the Table Key(s) and how should I handle them?”*”
- “*D.6 How do I configure the Primary Key field(s) of a Table?”*”.

The **AutoNumber** field type is **equivalent** to the VBA **Long** data type (click *G.2.1*).

#### D.4.11 What is the “OLE Object” field type?

The **OLE Object** field type stores **one OLE**<sup>21</sup> **object**. If you double-click on the field, MS-Access will **open** the OLE object using the default application to manage it.

Storing **one OLE object** in a field is **very useful** to immediately access a document associated with a record (e.g., the actual contract document corresponding to a contract record), but you risk your database file becoming **very large**, because the **OLE objects** are stored **inside** the MS-Access file.

If you **delete** the field content, you actually **delete** the **OLE object**, and it is **completely gone** (no paper trash recovery). If you **just deleted** the OLE object by mistake, you can recover it by clicking on the undo “” icon or pressing “**Ctrl-z**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**z**” key). If you **formerly deleted** the **OLE object** by mistake, and you want to recover it, you have to recover it from the backup of your database file.

Two useful alternatives to storing an **OLE Object** in a field are:

- Storing an **OLE Object** containing a **windows shortcut** to a file containing the actual OLE Object that you wanted to store. Notice that you have to **manually** create each specific windows shortcut.
- Storing a web **hyperlink** (click *D.4.12*) to a file containing the actual OLE Object that you wanted to store.

With either alternative above, your database file will not grow with the number of stored OLE Objects, but if you **move** the files containing the corresponding OLE Objects, the shortcuts/hyperlinks in the database are **not automatically updated**. You have to

---

<sup>21</sup> OLE stands for “Object Linking and Embedding”. If you want to know more, you may check: [https://en.wikipedia.org/wiki/Object\\_Linking\\_and\\_Embedding](https://en.wikipedia.org/wiki/Object_Linking_and_Embedding)

**manually update** the shortcuts/hyperlinks or rather you will not be able to directly access the OLE Objects from the corresponding fields.

The main difference between the **OLE Object** field type and the **Attachment** (click *D.4.13*) field type is that **OLE Object** can store only one OLE Object while **Attachment** can store several files.

The characteristics of the **OLE Object** field type are equivalent to the ones of the VBA *Object* data type (click *G.2.1*).


#### **D.4.12 What is the “Hyperlink” field type?**

The **Hyperlink** field type stores one web hyperlink. If you **click** inside the field, MS-Access will automatically **open** the hyperlink in the web navigator. Notice that if you click inside the field when the mouse has become a white cross, this selects **the field** (click *B.5.2*) instead of opening the hyperlink in the web navigator.

Storing one hyperlink in a field is **very useful** to immediately access a document/object associated with a record (e.g., the actual contract document corresponding to a contract record), and this does not add significant size to your database file because the actual document/object is **stored** elsewhere, and not in the database file. However, you have a risk of data incoherence if the actual document/object is **moved**, so the hyperlink stored in the field does not work.

If you want to **select** the hyperlink (instead of **opening** the hyperlink in the navigator), for example to delete or edit the hyperlink, you can do it in the following ways:

- Selecting **the field** by clicking inside the field when the mouse is near its border, once the mouse pointer has become a white cross (click *B.5.2*).
- Right-click inside the field and then press the “**Esc**” key.

If you **delete** the field content, you only **delete the stored hyperlink**, and the actual object pointed by hyperlink remains unaltered. If you **just deleted** the stored hyperlink by mistake, you can recover it by clicking on the undo “” icon or pressing “**Ctrl-z**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**z**” key). If you **formerly deleted** the stored hyperlink by mistake, and you want to recover it, you have to recover it from the backup of your database file.


The characteristics of the **Hyperlink** field type are equivalent to the ones of the VBA *String* data type (click *G.2.1*).

#### **D.4.13 What is the “Attachment” field type?**

The **Attachment** field type stores **one or more**, files of **any type**. If you double-click on the field, MS-Access will show a dialogue-box listing the files contained in the field. If you double-click on any of the file names, MS-Access will **open** the file using the default application associated to the file extension.

Storing **one or more** files in a field is **very useful** to immediately access the documents associated with a record (e.g., the different registration documents corresponding to a house record), but you risk your database file becoming **very large**, because all the attached files are stored **inside** the MS-Access file.

If you **delete** the field content, you actually **delete all the attached files**, and they are **completely gone** (no paper trash recovery). If you **just deleted** the attached files by

mistake, you can recover them by clicking on the undo “” icon or pressing “**Ctrl-z**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**z**” key). If you **formerly deleted** the attached files by mistake, and you want to recover them, you have to recover them from the backup of your database file.

Two useful alternatives to attaching files in a field are:

- Storing an **OLE Object** (click *D.4.11*) containing a **windows shortcut** to the actual file. Notice that you have to **manually** create each specific windows shortcut.
- Storing a web **hyperlink** to the actual file (click *D.4.12*).

With either alternative above, your database file will not grow with the number of stored files, but if you **move** the actual files, the shortcuts/hyperlinks in the database are **not automatically updated**. You have to **manually update** the shortcuts/hyperlinks or rather you will not be able to directly access the file from the corresponding fields.

The main difference between the **Attachment** field type and the **OLE Object** (click *D.4.11*) field type is that **Attachment** can store **several** files while **OLE Object** can store only **one** OLE Object.

#### **D.4.14 What is the “Lookup Wizard...”?**

This is **not** a **field type**. This is rather an assistant function that MS-Access provides to help you set up drop-down menus in fields, intended to simplify entering data into them.

My advice is **do not use this assistant**. Rather, I recommend using directly the “**Lookup**” tab in the bottom pane of the “Table pane” in “**Design View**”. You may check how to use the “**Lookup**” pane in “*D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?*”.

### **D.5 How do I configure a Table field validation rule, indexing, and other properties?**

Open the Table in “**Design View**” (click *B.4.1.3*).

At the top part of the “Table pane” you see the Table fields, one row for each field. When you click on the row of a field, it is selected, and you will see in the bottom part of the “Table pane” the **properties** of **this field**, in two tabs: “**General**” and “**Lookup**”. Selecting the “**General**” tab you see the **relevant properties** of the field that MS-Access allows you to configure. **Each property** is displayed as **one row** inside the “**General**” tab. The field properties depend on the **data type** of the field. There are some **common** properties to all data types, and some **specific** properties of each data type.

You may click:

- “*D.5.1 What common Table field properties should I set?*”
- “*D.5.2 What data-type specific Table field properties should I set?*”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

## D.5.1 What common Table field properties should I set?

I will explain, one by one, what I consider the **most relevant** field properties that are **common to most** data types, which are:

- **Description**  
Click “D.5.1.1 What is the “Description” Table field property?”.
- **Format**  
Click “D.5.1.2 What is the “Format” Table field property?”.
- **Caption**  
Click “D.5.1.3 What is the “Caption” Table field property?”.
- **Default Value**  
Click “D.5.1.4 What is the “Default Value” Table field property?”.
- (field) **Validation Rule**  
Click “D.5.1.5 What is the field “Validation Rule” Table field property?”.
- (field) **Validation Text**  
Click “D.5.1.6 What is the field “Validation Text” Table field property?”.
- **Required**  
Click “D.5.1.7 What is the “Required” Table field property?”.
- **Indexed**  
Click “D.5.1.8 What is the “Indexed” Table field property?”.
- **Text Align**  
Click “D.5.1.9 What is the “Text Align” Table field property?”.

Once you are done with your Table configuration, **save** (click B.4.1.6) your Table design. You may then **close** the Table (click B.4.1.7) or change it to “**Datasheet View**” (click B.4.1.4). If you get warning and/or **error** messages when saving your Table design, you may click “L.2 How do I fix errors with my Table/Form design?”.

### D.5.1.1 What is the “Description” Table field property?

This allows you to explain in some detail what is the meaning of each field. The field description is shown in the bottom frame of MS-Access when you select the Table field. This is very useful to show the user a detailed description of the field’s meaning.


To write the field description you should click on the box placed on the row of the field, and under the column labeled “**Description (Optional)**”. After having clicked there, you type-in the field description you want.

When you begin creating Table fields, you think the field name is enough to clearly explain the meaning of the field. However, when you advance with the design and with the usage of your database, you realize that the meaning of each field has to be explained with substantial detail. You should therefore write a field description in most of your fields.

### D.5.1.2 What is the “Format” Table field property?

This allows you to configure the format in which the value will be **shown** in the



“**Datasheet View**” of the Table. Available predefined formats from the drop-down menu of the “**Format**” property (click on the “” icon at the rightmost side of this property row) are quite self-explanatory. Important to point-out that you can configure a predefined format (by typing it in), even **if it is not shown** in the drop-down menu.

If you want to know more about predefined formats, you may click:

- “*H.6.1 How do I configure predefined column formatting in a Table/Query/Form?*”

If you want to configure custom formatting, you may click:

- “*H.6.2 How do I configure custom column formatting in a Table/Query/Form?*”


### **D.5.1.3 What is the “Caption” Table field property?**

This is the text **label** displayed at the **top** of this **column** in “**Datasheet View**”. If this is blank, MS-Access will display the **field name** as the text label of the column.

This feature is **extremely useful** in case you want to change the column label(s) of the Table, without having to modify the Queries. For example, if you want to translate all the database column labels to another language, you can do it in this way while keeping the **internal** field names of Tables and Queries in your own language.

### **D.5.1.4 What is the “Default Value” Table field property?**

This is the **default value** for this field. This value will appear **already typed-in** in the last row of each Table (the record with an asterisk “\*” to its left, which is used to type-in new records). This is quite useful to avoid typing the most usual value for the field. The default value is indicated by writing an **expression**. For example, a very frequent default value for fields of type **Date/Time** is the VBA function “**Date()**” that returns today’s date.

If you are familiar with the expression syntax in Table “**Design View**” (recall that MS-Access has **three** different expression syntax, click *G.1*), you may directly type-in the expression in the property “**Expression**”, in the field properties, placed at the bottom of the “Table pane”, in the tab “**General**”. If you are not familiar with the syntax of these expressions, you click on the icon with three dots “” at the right of this row, and MS-Access will open a dialogue-box to help you write the expression you want.

In the **expression** of the **default** value you can use constants, built-in operators, and built-in VBA functions. However, you **cannot** use any field name, nor user-defined VBA functions, nor Subqueries. If you want to know more about the rules to write expressions in Table “**Design View**”, you may click “*G.1 What are the main differences between the three expression scopes?*”.

### **D.5.1.5 What is the field “Validation Rule” Table field property?**

It is a **Boolean expression** that **must not** return **False** for any **new** value for **this field**. If the **Boolean** expression returns **Null**, the field validation rule is considered correct.

A field validation rule can for example require that a number is greater than or equal to zero, or less than ten thousand, or more complex requirements using built-in functions. A couple examples of simple expressions that you could use as field validation rules

are:

```
[field_name] > 0  
([field_name] > 0) AND ([field_name] < 10000)
```

Notice that **Null** in “field\_name” **satisfies** the two field validation rules above.

Configuring a field validation rule prevents **both** introducing **new** records violating the field validation rule and changing this field’s value in an **existing** record such that the value violates the field validation rule. However, if the Table **already** contains records that violate this field validation rule, the records **will stay** in the Table **after** you configured the field validation rule. Notice that you can edit **other fields** of an **existing** record that violates this field validation rule, and the resulting edited record that still violates this field validation rule will remain in the Table.

This is very **important**, because if you see a field validation rule, you should remind that the field **can contain values that violate it** in records existing in the Table **before** the field validation rule was configured.

If you configure a field validation rule in a Table with existing records, when saving the Table design MS-Access will ask if it should check if all the values **existing** in the Table comply with the field validation rule (click *L.2.1*).

Be **well aware** that if the **Boolean** expression (in the field validation rule) returns **Null**, the field validation rule is **correct**, and the value is **accepted** in the field. The rationale for this is that **Null** can only appear in “optional” fields, this is, fields with optional values that are configured as “**Required=No**”. At the same time, if you have an optional field containing **Null**, this most likely causes the **Boolean** expression (in the field validation rule) to return **Null**. In this case, you **usually** want the field value to be **accepted**, because you accept **Null** in optional fields.

If you want to check for records with **field values that violate the field validation rule** in records already existing in the Table before you configured the field validation rule, you remove the field validation rule, then configure the field validation rule again, and save the Table design (click *B.4.1.6*). You then click on “**Yes**” upon the question of testing all existing data (click *L.2.1*).

If you want to **change** field values that violate the field validation rule already existing in the Table **before** you configured the field validation rule, you may click “*E.7 How do I bulk-change my Table/Form’s data?*”.

Field validation rules are **extremely useful** to avoid errors in your database, so I **strongly recommend you use them** for **every** field where there are known mandatory restrictions in its values.

### **How do I write the Boolean expression in the field validation rule?**

If you are familiar with the expression syntax in Table “**Design View**” (recall that MS-Access has **three** different expression syntax, click *G.1*), you may directly type-in the expression in the property “**Expression**”, in the field properties, placed at the bottom of the “Table pane”, in the tab “**General**”. If you are not familiar with the syntax of these expressions, you click on the icon with three dots “**...**” at the right of this row, and MS-Access will open a dialogue-box to help you write the expression you want.

In the **Boolean** expression of a field validation rule you **can use** constants, built-in operators, built-in VBA functions and the field’s name. However, you **cannot use** any

other field names of the Table, nor user-defined VBA functions, nor Subqueries. If you want to know more about the rules to write expressions in Table “**Design View**” you may click “*G.1 What are the main differences between the three expression scopes?*”.

Two examples of frequently used field validation rules are enforcing that a **Date/Time** value has zero-date (this is, the date is December 30th 1899, equivalent to integer part equal zero), as if it had no date information or enforcing that it has the time 0:00:00 (fractional part zero) as if it had no time information. These validation rules are:

- Rule for zero-date: `Fix([field_name]) = 0`
- Rule for zero-time: `Fix([field_name]) = [field_name]`

#### **D.5.1.6 What is the field “Validation Text” Table field property?**

This is the text that MS-Access will show you when you type-in a data value that returns **False** in the expression that you wrote in the “**Validation Rule**”. If you leave this box blank, MS-Access will show a generic message indicating that the value does not fulfil the validation expression and showing the validation expression. Seeing the validation expression is quite self-explanatory in some cases, but for non-expert users or in case you have a non-obvious validation rule, using this field will be **very** useful to explain what the value restrictions in this field are. My advice is you always use this property. To use it, just type-in the validation text you want MS-Access to show.

#### **D.5.1.7 What is the “Required” Table field property?**

If you configure a Table field as “**Required=Yes**”, MS-Access will **not** allow you to introduce any **new Null** in this field.

Configuring “**Required=Yes**” prevents **both** introducing **new** records having **Null** in this field and changing this field to **Null** in an **existing** record. However, if the Table **already** contains records with **Null** in this field, the records with **Nulls will stay** in the Table **after** you configured “**Required=Yes**”. Notice that you edit **other fields** of an **existing** record having **Null** in this field, and the resulting edited record with **Null** in this field will remain in the Table.

This is very **important**, because if you see a field configured as “**Required=Yes**”, you should remind that the field **can contain Nulls** in records existing in the Table **before** the field was configured as “**Required=Yes**”.

If you configure a field as “**Required=Yes**” in a Table with existing records, when saving the Table design MS-Access will ask if it should check if all the values **existing** in the Table comply with “**Required=Yes**” (click *L.2.1*).

Allowing **Nulls** in Table fields is **very undesirable** for several reasons (click *C.6, 0, K.1.3* and *K.5*). My advice is to **restrict very much** the fields that you configure as “**Required=No**”, and invest a little design effort to have **all**, or **almost all**, your fields configured as “**Required=Yes**”.

If you want to check for records with **Nulls** in a field, you configure the field as “**Required=No**”, then configure it as “**Required=Yes**”, and save the Table design (click *B.4.1.6*). You then click on “**Yes**” upon the question of testing all existing data (click *L.2.1*).

If you want to **remove Nulls** in a field in records already existing in the Table **before**

you configured the field as “**Required=Yes**”, you may click “*E.7 How do I bulk-change my Table/Form’s data?*”.

#### **D.5.1.8 What is the “Indexed” Table field property?**

Indexing is **really valuable** in your database (click “*C.8 What is indexing?*”), so you should set it as much as it makes sense (quite frequently). If you click on the rightmost side of the “**Indexed**” row, you will get a drop-down menu where you may choose the values:

- “No”
- “Yes (Duplicates OK)”
- “Yes (No duplicates)”

If you **are sure** that this field **must not have duplicate** values, configure “**Indexed=Yes (No duplicates)**”.

If you **are sure** that this field can have **duplicates** and **will not** be used in any “**WHERE**”, “**ON**” or “**GROUP BY**” expressions, configure “**Indexed=No**”.

If you are not sure about the two conditions I just indicated, configure “**Indexed=Yes (Duplicates OK)**”.

If you configure a field as “**Indexed=Yes (No duplicates)**” in a Table with existing records, when saving the Table design MS-Access will **check** if all the values **existing** in the Table are duplicate-free in this field. If there is **any** duplicate, MS-Access will not allow you to configure “**Indexed=Yes (No duplicates)**” (click *L.2.6*). Remind that for this purpose, having several **Null** is not considered as duplicate values (click *C.8.3.2*).

#### **D.5.1.9 What is the “Text Align” Table field property?**

This allows you to select the alignment with which MS-Access will show the values of this field in “**Datasheet View**”. Click on the **drop-down menu** “” icon at the rightmost side of this row, and you can then select one of the options. The default option is “**General**” in which **numbers** and **dates** are **right justified** while **text** is **left justified**.

### **D.5.2 What data-type specific Table field properties should I set?**

In addition to the **common** field properties (click *D.5.1*), you may see **some other** Table field properties that are **specific** to some data types. I will explain the following ones because I think they are quite important (you may click on the one you want):

- “*D.5.2.1 What is the “Field Size” Table field property?*”
- “*D.5.2.2 What is the “Allow Zero Length” Table field property?*”
- “*D.5.2.3 What is the “Decimal Places” Table field property?*”
- “*D.5.2.4 What is the “Input Mask” Table field property?*”
- “*D.5.2.5 What is the “Result Type” Table field property?*”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

### **D.5.2.1 What is the “Field Size” Table field property?**

**This property is available for field types: Short Text, Number and AutoNumber.**

Its functionality depends on the field type, as follows:

- **In a Short Text field**

The “**Field Size**” property is the maximum number of characters that this Table field can contain. My advice is you always configure it to the default maximum length of 255. You may click *D.4.1*.

- **In a Number field**

The “**Field Size**” property is a refinement of the data type of the field, as I have explained in “*D.4.3 What are the “Number” field types?*”.

- **In an AutoNumber field**

The “**Field Size**” property can take the values “**Long Integer**” or “**Replication ID**”.

### **D.5.2.2 What is the “Allow Zero Length” Table field property?**

**This property is only available for field types: Short Text and Long Text.**

If you configure “**Allow Zero Length=No**”, MS-Access will require that you input at least one non-blank character in this field. If you configure it to “**Yes**”, MS-Access will allow you to input text strings of zero characters (remind that a text string with zero characters is **different** from a **Null!!!**).

My advice is you **always** set this property to “**No**”. The reason is the **zero-length** string is **not as bad as Null**, but it still has a **significant risk** of causing errors because it can be mistaken with **Nulls** and with invisible strings (click *L.7.8*). In case you want to know more about these problems, you may click “*L.7 How do I fix errors with Short Text or String fields?*”.

If you configure a field as “**Allow Zero Length=No**” in a Table with existing records, when saving the Table design MS-Access will ask if it should check if all the values **existing** in the Table field comply with “**Allow Zero Length=No**” (click *L.2.1*). However, even if you answer “**Yes**”, MS-Access will not check it! This may be an MS-Access **bug**.

### **D.5.2.3 What is the “Decimal Places” Table field property?**

**This property is only available for field types: Number, Currency and Large Number**

This is the number of **decimal digits** that will be **shown** in “**Datasheet View**”. Notice that this is just a **visualization** issue, because the **actual value** will be stored in the field with as many decimals allowed by the underlying data type (simple or double precision).

### **D.5.2.4 What is the “Input Mask” Table field property?**

**This property is only available for field types: Short Text and Date/Time.**

This allows you to restrict the values typed-in into this field to the ones that satisfy a certain format. MS-Access offers some built-in input masks that you can select, and also, it allows you to define your own input masks.

For **Date/Time** fields MS-Access offers some built-in input masks to enter zero-time or

zero-date.

For **Short Text** fields MS-Access offers some built-in input masks.

On my view, an input mask is not very useful, because it only restricts what you can **type-in**, but it **does not** restrict the actual values that the field can contain. If you paste values, the values **will not** be checked by the input mask. Therefore, the input mask just introduces **restrictions** in the way you can **type-in** your values but it **does not** protect you from erroneous values, because you can still paste them into the fields.

My advice is that you avoid using the “**Input Mask**” property and configure instead the field “**Validation Rule**” property (e.g., to enforce zero-time or zero-date values). Check my explanation about the field “**Validation Rule**” property in “*D.5.1.5 What is the field “Validation Rule” Table field property?*”.

#### **D.5.2.5 What is the “Result Type” Table field property?**

**This property is only available for field type Calculated.**

This allows you to configure, in a combined way, the field type and the field size of this field. If you want to know more about the **Calculated** field type, you may click “*D.4.8 What is the “Calculated” field type?*”.

Notice that depending on the “**Result Type**” that you configure, MS-Access will show different properties for this field, both in the “**General**” and in the “**Lookup**” tabs.

### **D.6 How do I configure the Primary Key field(s) of a Table?**


You may click:


- “*D.6.1 How do I configure a Primary Key with only one field?*”
- “*D.6.2 How do I configure a Primary Key with several fields?*”
- “*D.6.3 How do I change the Primary Key of a Table?*”
- “*D.6.4 What checks are done on Key fields when saving my Table design?*”
- “*D.6.5 How do I identify the Primary Key field(s) of a Table?*”
- “*D.6.6 Why should I define the Primary Key field(s) in Tables?*”

#### **D.6.1 How do I configure a Primary Key with only one field?**

Open the Table in “**Design View**” (click *B.4.1.3*).

You configure **only one field** as the **Primary Key** of the Table (this is called a “**simple Primary Key**”), in either of the following ways:

- Right-click anywhere on the **field row** and click on “**Primary Key**” from the pop-up menu.
- Click anywhere on the **field row** and click on the **Primary Key** “” icon from the “**Table Tools / Design**” contextual Ribbon.

Regardless of how you did it, you should now see a key “” icon just to the left of the **field row**: this indicates it has been configured as the Table’s (simple) **Primary Key**.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table

design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).


In case the Table **already has** one or more **records**, when you **try** to **save** the Table design, MS-Access will **check** that the values of the **Primary Key field** in all the Table records have no duplicate values and also that the **Primary Key field** contains no **Nulls**. If you want to know more about the possible results of this value check, you may click “*D.6.4 What checks are done on Key fields when saving my Table design?*”.


You can also configure the **Primary Key** using the “**Indexes**” dialog box (click *D.7.2*).

### **D.6.2 How do I configure a Primary Key with several fields?**

Open the Table in “**Design View**” (click *B.4.1.3*).

Select the **field rows** (click *B.6.1.4*) you want to become the **Primary Key**.

Click on the **Primary Key** “” icon from the “**Table Tools / Design**” contextual Ribbon.

You should now see a key “” icon just to the left of **each field row**: this indicates that **all** of them have been **jointly** configured as the Table’s (composite) **Primary Key**.

If you configure a **composite Primary Key**, I advise you **also** configure “**Indexed=Yes (With duplicates)**” in **each and every field** of a composite **Primary Key**. This will **improve the performance** of your Queries and other operations over the database.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

In case the Table **already has** one or more **records**, when you **try** to **save** the Table, MS-Access will **check** that the values of the **Primary Key fields** in all the Table records have **no combined duplicate values** and also that the **Primary Key fields** contain no **Nulls**. If you want to know more about the possible results of this value check, you may click “*D.6.4 What checks are done on Key fields when saving my Table design?*”.


You can also configure the **Primary Key** using the “**Indexes**” dialog box (click *D.7.2*).


### **D.6.3 How do I change the Primary Key of a Table?**

Open the Table in “**Design View**” (click *B.4.1.3*).

You **replace** the current **Primary Key** by a new one by just configuring the **new Primary Key** fields that you want: click *D.6.1* for one field (simple **Primary Key**) and *D.6.2* for several fields (composite **Primary Key**). Configuring the **new Primary Key** field(s) will also **clear** the **old Primary Key** fields.

If you rather want to **clear** the **Primary Key** fields, and leave this Table **without a Primary Key**, you can do it in either of the following ways:

- Right-click anywhere on **any** of the **field rows** of the current **Primary Key** and click on “**Primary Key**” from the pop-up menu.
- Click anywhere on the **field row** of any of the field rows of the current **Primary Key** and click on the **Primary Key** “” icon from the “**Table Tools / Design**” contextual Ribbon.

Regardless of how you did it, **all** the key “” icons that were placed on the left side of the **field rows** of the former **Primary Key** fields show now be **gone**, and **no field** should have such a key icon.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

In case the Table **already has** one or more **records**, when you **try** to **save** the Table, MS-Access will **check** that the values of the **Primary Key fields** in all the Table records have **no combined duplicate values** and also that the **Primary Key fields** contain no **Nulls**. If you want to know more about the possible results of this value check, you may click “*D.6.4 What checks are done on Key fields when saving my Table design?”.*

You can also configure the **Primary Key** using the “**Indexes**” dialog box (click *D.7.2*).

#### **D.6.4 What checks are done on Key fields when saving my Table design?**

If you configure the Table **Primary Key** field(s), and the Table already has one or more records, when you **try** to **save** the Table, MS-Access will **check** that the values of the **Key** field(s) in all the Table records have no (combined) duplicate values and also that the **Key** field(s) contain no **Nulls**.

- If the **check** is not passed because any of the **Key** field(s) contains one or more **Null**, MS-Access will **not** save the Table and will show the error message:  
“*Index or primary key cannot contain a Null value.*”

If you want more information on this, you may click “*L.2.7 How do I fix “...primary key cannot contain a Null...” when saving my Table design?”.*”

- If the **check** is not passed because there are one (or more) records with duplicate values in the **Key** field(s), MS-Access will **not** save the Table and will show the error message:  
“*The changes you requested to the table were not successful because they would create duplicate values in the index, primary key, or relationship. Change the data in the field or fields that contain duplicate data, remove the index, or redefine the index to permit duplicate entries and try again.*”

If you want more information on this, you may click “*L.2.6 How do I fix “Key/Index with duplicate values” when saving my Table design?”.*”

- If the **check** is passed, the Table is saved with the **Key** fields you wanted properly configured.

If you get **other** warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?”.*”

#### **D.6.5 How do I identify the Primary Key field(s) of a Table?**

You first identify the **candidate** Key(s) of the Table (click *C.10.3*).

In case there is **only one candidate Key** it is clear that **it must be the Primary Key** of the Table. However, if a Table happens to have **more than one candidate Key**, you should carefully decide which one is **the Primary Key** of the Table.



Let me show you this with an example. Imagine you have a Table of capital cities with only two fields “Capital” and “Country”. You want all records to have their values of “Capital” and “Country”, so you configure **both** as “**Required=Yes**” (i.e., as being **without Nulls**). Since there can be no duplicate values neither on “Capital”, nor on “Country”, to avoid mistakes you have configured **each** of both fields as indexed **without** duplicate values. The Table could look like:

T_Capitals	
Capital	Country
Beijing	China
Brasilia	Brazil
Buenos Aires	Argentina
Madrid	Spain
Washington	United States

You probably have already noticed that **both fields** “Capital” and “Country” are a **simple candidate Key** for this Table. The decision of which of both fields is **the** Table **Key** is subtle, but relevant. You have to decide if this is a Table of **countries**, with an additional field to indicate the capital of the country, or rather, this is a Table of **capital cities**, with an additional field to indicate the country of each capital city. Without further information about the database where this Table is included it is not possible to know what of both options is correct. Let us therefore imagine that you also have in the Table a field “Inhabitants”. The decision is now pretty clear: if the field “Inhabitants” contains the **city** inhabitants, then “Capital” should be the **Key** field. However, if “Inhabitants” contains the **country** inhabitants, then “Country” should be the **Key** field.

Let me present you one more example. Imagine that you have a Table of quarterly rain measurements in capital cities, as the example Table from chapter A.5:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

**Each** record in this Table represents an **entity** which is **one quarterly** measurement of rainfall **in one capital city**. As I have indicated, the **Key** field(s) are the ones that “**identify**” each **entity** and each **unique** record in the Table. Therefore, the **Key** fields cannot have duplicate values in every **current or future** record in the Table. The field “Capital” is not the **Key**, because you can have several records with the same “Capital” value. The same happens for all the other fields in the Table: **none** of them has unique values across the whole Table, so you **do not** have **any simple candidate Key**.

However, if you think about it, what should be **unique** is the **combined** value of “Capital”, “Cal\_Year” and “Quart”. This is what identifies each valid measurement record in this Table. Each of these unique records will have the corresponding value in the field “Quart\_Rainfall”, which represents the amount of rain that the capital in the record got in that quarter from that year. Therefore, each entity (a quarterly rainfall measurement in a Capital) is identified by the composite values of “Capital”, “Cal\_Year” and “Quart”, because these three fields identify **one and only one** measurement, and therefore, identify one and only one record. The fourth field “Quart\_rainfall” is providing one **attribute** to this entity. The attribute is the value of the rainfall in that specific quarter, in that specific capital city. Therefore, this Table has a composite **Key**, and the **Primary Key** fields you should configure are “Capital”, “Cal\_Year” and “Quart”.

As a final remark on identifying the **Key** fields, remind that **Key** fields **cannot be empty**, this is, **none** of the **Key** fields **can contain a Null**. If you try to define Key field(s) that have a **Null** in one (or more) records, MS-Access will not allow you, and will give you an informative message. If you have that problem, the solution is quite simple: you first replace all the currently **Null** elements in all the **Key** field(s) with a correct value, and then MS-Access will allow you to select them as the **Key** field(s).

### **D.6.6 Why should I define the Primary Key field(s) in Tables?**

Defining the **Primary Key** field(s) of your Tables allows the system to uniquely identify each record contained in the Table, prevents duplicate records and it is a **best practice** in database design.

In the most frequent case where you have **only one candidate Key**, defining the **primary Key** field(s) of the Table is trivial, because you just need to configure the **only** set of candidate **Key** field(s) as the **primary Key** field(s). If you define **one** field, or **one** group of fields, as the **primary Key** field(s), MS-Access will automatically configure an index without duplicate values and without **Nulls**. Therefore, you do not need to configure **first** an index over them, and **then** configure them as the **Primary Key** fields: just configure them as the **Primary Key** and the index is automatically configured.

In the very few cases where you have more than one **candidate Key**, you only need to think which one should be **the Primary Key**. This small extra effort really pays off. Actually, the usual way to configure this in MS-Access is by **configuring first** the **Primary Key** field(s), and then, **optionally** configuring one (or more) index(es) with or without duplicate values with or without **Nulls**.

### **D.7 How do I add simple and/or composite index(es) to a Table?**

You may click:

- [\*“D.7.1 How do I add simple indexes to a Table?”\*](#)
- [\*“D.7.2 How do I add composite \(and simple\) indexes to a Table?”\*](#)
- [\*“D.7.3 How do I reconfigure the indexes of a Table?”\*](#)


### D.7.1 How do I add simple indexes to a Table?

To configure **simple** indexes, you can do it when configuring each Table field: click *D.5.1.8* to configure its “**Indexing**” property. If you want the simple index to be **without Nulls**, click *D.5.1.7* to configure “**Required=Yes**”.

You can **also** configure **simple** indexes using the “**Indexes**” tool, that is used to configure **composite** indexes. If you want to do this, you may click “*D.7.2 How do I add composite (and simple) indexes to a Table?*”.

### D.7.2 How do I add composite (and simple) indexes to a Table?

Open the Table in “**Design View**” (click *B.4.1.3*).

Click on “**Design**” from the “**Ribbon-bar**” (below the “**Table Tools**” contextual label) and then click on the **Indexes** “” icon. This opens a dialog box called “**Indexes**” that shows **all** the indexes in the Table and allows to **create new** indexes and/or to **configure existing** ones.

If you had configured indexing on some individual fields (click *D.5.1.8*) and/or a simple or composite **Primary Key** (*D.6*), the corresponding indexes will be **shown** in the “**Indexes**” dialog box (if you want to manage them, you may click *D.7.3*).

To **add** a **new** index, find a **blank row**, and on the cell in the column “**Index Name**” write an index name that **does not already exist** in the “**Indexes**” dialog box. The index names are **not** used externally, and they are **only** used within the “**Indexes**” dialog box to tell apart the different indexes defined in this Table.

Then click on the rightmost side of the cell to its right (under the column “**Field Name**”) and click on a Table field from the drop-down menu that will be shown. You have now configured an index with one field (a **simple** index). If you want a **composite** index, you have to configure **additional** field(s) in this index. You do this by clicking on the rightmost side of the cell **below** this one (also under the column “**Field Name**”) and click on **another** Table field from the drop-down menu that will be shown. Do the same process with as many fields you want to add fields to this **composite index** (up to a maximum of 10 fields). If you want, you may configure the “**Sort Order**” of each field, although my advice is to leave always the default “**Ascending**” value.

Notice that the fields that make a **composite index** are ordered, and the order you choose has an impact on the performance of the index. If you want to improve performance, put first the fields with less duplicate values and last the fields with more duplicate values.

Notice also that you can add to an index the **same** field **several** times, although this is not very useful.

Once you have defined the field(s) in your **new index**, click on the row that contains the index name, and this will show its **three** properties in the bottom part of the “**Indexes**” dialog box. The three properties shown are:

- “**Primary**”

If you set “**Primary=Yes**”, then you are configuring these index’s field(s) as the actual **Primary Key field(s)** of the Table. If you set “**Primary=No**”, then this index’s fields **are not** the **Primary Key field(s)** of the Table.

- “**Unique**”

If you set “**Unique=Yes**”, this index does **not allow duplicate values**. If you set “**Unique=No**”, this index **allows duplicate values**. Remind that preventing duplicate values does not imply preventing duplicate records. For the latter requirement you also need to configure **all** the fields in the index as “**Required=Yes**” (click *D.5.1.7*).

- “**Ignore Nulls**”

If you set “**Ignore Nulls=Yes**”, any record having **Null** in an index field **will not be included** in this index. If you set “**Ignore Nulls=No**”, all the Table records will be included in this index. If all the fields in the index are configured as “**Required=Yes**” (i.e., all of them are **without Nulls**), the value of “**Ignore Nulls**” is irrelevant, because the fields cannot contain **Null**. If this index has one or more fields configured as “**Required=No**”, my advice is you set “**Ignore Nulls=Yes**” in case you expect lots of **Null** (say more than 30%) in one or more of the fields configured with “**Required=No**”.

Notice it is very different to configure an index to **ignore** records that contain **Null** in any of the index fields, than configuring all the fields in the index to be **without Nulls** (i.e., configure “**Required=Yes**”). If you want to know more about this, you may click “*C.8.4 What indexes can I configure in a given Table?*”

You can create as many **indexes** as you need (subject to the maximum capacity of MS-Access).

Once you are done configuring your index(es), **close** the “**Indexes**” dialog box by clicking on the close “**X**” icon on its top-right corner.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”. You may finally **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

### **D.7.3 How do I reconfigure the indexes of a Table?**

Open the “**Indexes**” dialog box (click *B.6.2.1*).

You **delete an index** by **deleting all** its **index field rows** (click *B.6.2.3*). Notice that if you **do not** delete **all** of them, you will get a different result:

- If you deleted the **first** index field row of the index, then the non-deleted index field rows **will become part** of the index that is placed right above this one.
- If you **did not** delete the **first** index field row of the index, then this index is **not deleted**, and you just deleted some of its fields.

Both results above are obviously bad if they were not intentional, so make sure you delete **exactly all** the **index field rows** of the index you want to delete, no more, no less.

You **delete fields** from an index by **deleting** the corresponding **index field rows** (click *B.6.2.3*). In case the field you want to delete is the one on the **first** index field row of the index, then you **cannot** delete it because this would **delete the index itself**. In this case, you **reconfigure** the **first index field row** with the data of one of the index fields that you **do not** want to delete (i.e., you are duplicating this field), and then you

delete that index field row instead of the first one.

You **change** the **order** of the **fields** from an index by **moving** the **index field rows** (click [B.6.2.3](#)) as corresponds.

You **add a field** to an index, by **inserting** an **index field row** (click [B.6.2.3](#)) in the corresponding place within the index field rows, and then configuring it (click [D.7.2](#)). Notice that you can add to an index the **same field several** times, although this is not very useful.

You **reconfigure** an **index field row**, or the **index itself**, by modifying their corresponding **properties** (click [D.7.2](#)).

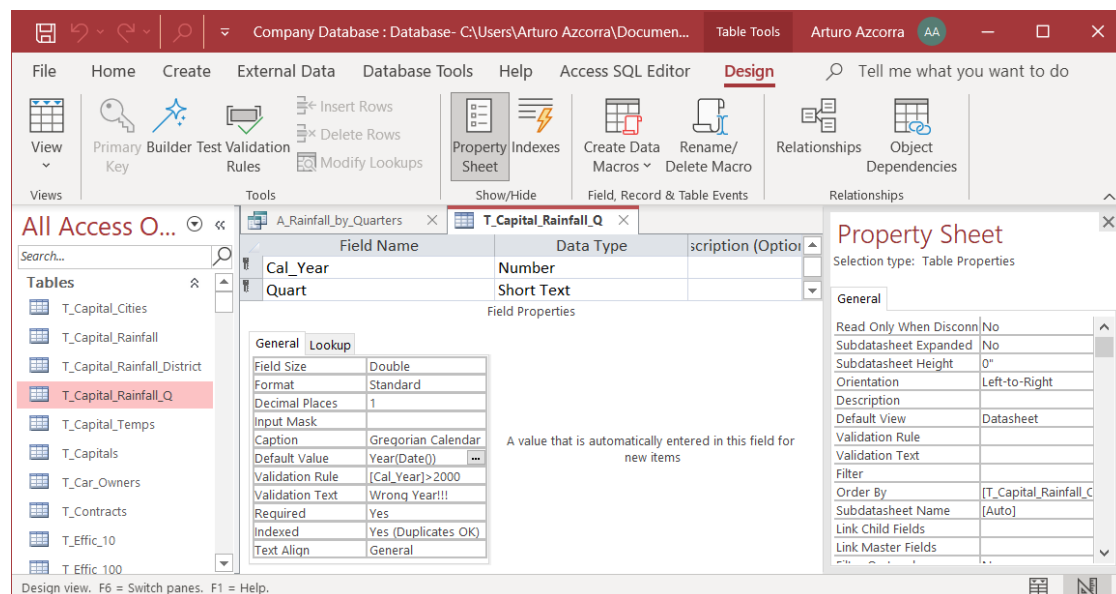
Once you are done reconfiguring your index(es), **close** the “**Indexes**” dialog box by clicking on the close “**X**” icon on its top-right corner.

Once you are done with your Table configuration, **save** (click [B.4.1.6](#)) your Table design. You may then **close** the Table (click [B.4.1.7](#)) or change it to “**Datasheet View**” (click [B.4.1.4](#)). If you get warning and/or **error** messages when saving your Table design, you may click “[L.2 How do I fix errors with my Table/Form design?](#)”.

## D.8 How do I configure the properties of a Table?

**Unhide** the “**Property Sheet**” of the Table (click [B.7.1](#)).

The following screenshot shows the “**Property Sheet**” of a Table, placed on the right side of the “Table pane”:



You can see that the “**Property Sheet**” of the Table only has one tab called “**General**”, and this tab contains a number of rows, with one property in each row. I will explain with more detail the following properties, that you may click:

- “[D.8.1 How do I configure a record validation rule?](#)”
- “[D.8.2 How do I configure the record validation text?](#)”
- “[D.8.3 How do I configure the “Subdatasheet Name” property? How do I configure the “Subdatasheet Name” property?](#)”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

### **D.8.1 How do I configure a record validation rule?**

A record validation rule is a **Boolean expression** that **must not** return **False** for the values of **any new record**. If the **Boolean** expression returns **Null**, the record validation rule is considered correct.

A record validation rule can for example require that one field is greater than or equal to another field, or that the sum of two fields is less than some given constant. An example of a simple expression that you could use as a record validation rule is:

```
[Begin] <= [End]
```

this record validation rule would **enforce** that the content of field “Begin” is less than or equal to the content of field “End” in any **newly inserted** record in the Table. Notice that **Null** in “Begin” and/or “End” **satisfies** this record validation rule.

Another example of record validation rule is:

```
([Total] >= [Item_1]) AND ([Total] >= [Item_2])
```

this record validation rule would **enforce** that the content of field “Total” is greater than or equal to the content of field “Item\_1”, and also, that the content of field “Total” is greater than or equal to the content of field “Item\_2” in any **newly inserted** record in the Table. Notice that **Null** in one (or more) of the fields **satisfies** this record validation rule.

Configuring a record validation rule prevents **both** introducing **new** records violating the field validation rule and changing the values of the fields (involved in the record validation rule) in an **existing** record such that their values violate the record validation rule. However, if the Table **already** contains records that violate the record validation rule, the records **will stay** in the Table **after** you configured the record validation rule. Notice that you can edit **other fields** (i.e., fields not included in the **Boolean** expression) of an **existing** record that violates the record validation rule, and the resulting edited record that still violates the record validation rule will remain in the Table.

This is very **important**, because if you see a record validation rule, you should remind that the Table **can contain records that violate it** if they existed in the Table **before** the record validation rule was configured.

When you configure a record validation rule in a Table with existing records, you will be asked if MS-Access should check if all the records **existing** in the Table comply with the record validation rule (click *L.2.1*).

Be **well aware** that if the **Boolean** expression (in the record validation rule) returns **Null**, the record validation rule is **correct**, and the record is **accepted** in the Table. The rationale for this is that **Null** can only appear in “optional” fields, this is, fields with optional values that are configured as “**Required=No**”. At the same time, if you have optional fields containing **Null**, this most likely causes the **Boolean** expression (in the record validation rule) to return **Null**. In this case, you **usually** want the record to be **accepted**, because you accept **Null** in optional fields. In any case, you can design the

**Boolean** expression to return **False** upon any combination of **Null** in the fields, as you want.

If you want to check for records **that violate the record validation rule** for records already existing in the Table before you configured the record validation rule, you remove the record validation rule, then configure the record validation rule again, and save the Table design (click *B.4.1.6*). You then click on “**Yes**” upon the question of testing all existing data (click *L.2.1*).

If you want to **change** field values in records that violate the record validation rule already existing in the Table **before** you configured the record validation rule, you may click “*E.7 How do I bulk-change my Table/Form’s data?*”.

Record validation rules are **extremely useful** to avoid errors in your database, so I **strongly recommend you use them** for **every** Table where there are known mandatory restrictions in its related field values.

### **How do I define the *Boolean* expression in the record validation rule?**

If you are familiar with the syntax of expressions in Table “**Design View**” (recall that MS-Access has three different expression syntax, click *G.1*), you may directly type-in the expression in the box to the right of “**Validation Rule**”. If you are not familiar with the syntax of these expressions, you click on the icon with three dots “**...**” at the right of this row, and MS-Access will open a dialogue-box to help you write the expression you want.

In the **Boolean** expression of a record validation rule you **can use** constants, built-in operators, built-in VBA functions and all the record’s field names. However, you **cannot use** user-defined VBA functions, nor Subqueries. If you want to know more about the rules to write expressions in Table “**Design View**” you may click “*G.1 What are the main differences between the three expression scopes?*”.

Record **validation rules** are **extremely useful** to avoid errors in your database, so I **strongly recommend you use them** for **every** Table where there are known mandatory restrictions in relation to its combined field values.


### **D.8.2 How do I configure the record validation text?**

This is the text that MS-Access will show you when you type-in a **record** that returns **False** in the expression you wrote in the record “**Validation Rule**” from the Table’s “**Property Sheet**”. If you leave this box blank, MS-Access will show a generic message indicating that the value does not fulfil the validation expression and showing the validation expression. Seeing the validation expression is quite self-explanatory in most cases, but in case you have a non-obvious validation expression, using this field will be very useful to explain what the joint value restrictions in the fields of this Table are. If you want to use this property, just type-in the validation Text you want MS-Access to show.

### **D.8.3 How do I configure the “Subdatasheet Name” property?**

A “**Subdatasheet**” is a **datasheet** (similar to an MS-Access object in “**Datasheet View**”) linked to records from **another datasheet**. This linking between **datasheets** may be repeated hierarchically up to a maximum of eight levels. MS-Access allows you to configure a **subdatasheet** in any Table, Form or Query results (all in “**Datasheet**

View”).

When a Table has an associated **subdatasheet**, a plus “+” icon will be displayed on the left side of each record. If you click on **one** of the plus “+” icons, it will change to a minus “-” icon, and the **subdatasheet** corresponding to that record will be **unhidden**. If you click on a minus “-” icon, it will change to a plus “+” icon, and the **subdatasheet** corresponding to that record will be **hidden**. You can also **unhide** or **hide** the **subdatasheets** corresponding to **all** the Table records by successively clicking on “**Home**”, on the more “” icon, on “**Subdatasheet**” from the pop-up menu, and finally on either “**Expand All**” or “**Collapse All**” (respectively) from the secondary pop-up menu.

For the case of a Table involved in a Relationship, MS-Access **automatically** configures the **slave** Table as a **subdatasheet** of the **master** Table. If a Table is a **master** Table in **several** Relationships, the first time that you click on one of its plus “+” icons MS-Access will ask you which of its **slave** Tables you want to configure as the **subdatasheet**.

You may **add**, **modify** or **remove subdatasheets** (including the ones automatically configured by MS-Access for the case of a Table Relationship) by configuring the Table properties, as follows:

**Open** the Table in “**Design View**” (click *B.4.1.3*) or change its **view-type** to “**Design View**” (click *B.4.1.4*).

**Unhide** the “**Property Sheet**” of the Table (click *B.7.1*).

You can then do the configuration(s) that you want:

- To **remove** the **subdatasheet**:  
Change the “**Subdatasheet Name**” property to “[None]”.
- To **keep** the default **subdatasheets**:  
Leave the “**Subdatasheet Name**” property with its default value of “[Auto]”. If you had changed the default value, then change it back to “[Auto]”.
- To **add** or **modify subdatasheets**:  
Configure the “**Subdatasheet Name**” property, as well as the related “**Link Child Fields**” and “**Link Master Fields**” properties.

I personally consider that **subdatasheets** have more **disadvantages** (risk of confusion by the user) than **advantages** (powerful data visualization tool). For this reason, I feel more inclined to remove all **subdatasheets**.

## **D.9 How do I create and configure my Table Relationships?**

You may click:

- “*D.9.1 How do I create a new Relationship?”*”
- “*D.9.2 How do I configure a Relationship?”*”
- “*D.9.3 What is the effect of “Enforce Referential Integrity”?”*”
- “*D.9.4 What is the effect of “Cascade Update Related Fields”?”*”
- “*D.9.5 What is the effect of “Cascade Delete Related Records”?”*”



- “D.9.6 What Relationships should I configure?”

## D.9.1 How do I create a new Relationship?

You may click:

- “D.9.1.1 How do I create a new Relationship in the most frequent case?”
- “D.9.1.2 How do I create a new Relationship in less frequent cases?”
- “L.3 How do I fix errors in my Relationship configuration?”

### D.9.1.1 How do I create a new Relationship in the most frequent case?

Open the “**Relationships**” pane (click *B.10.1*). The first time that you open it, the “**Relationships**” pane will be blank (with gray background).

**Unhide** both the **master** and the **slave** Table-boxes (click *B.10.4.4*) between which you want to establish the Relationship. This is **mandatory** because you can **only** create a Relationship between **Table-boxes** that are **visible** in the “**Relationships**” pane.

You can now create a **new** Relationship in either of the following ways:

- Drag-and-drop a **master field name** from the **master** Table-box to its **slave** field in the **slave** Table-box. Doing this will **create** the Relationship and open an “**Edit Relationships**” box. You now **configure** (click *D.9.2*) the just created Relationship using the “**Edit Relationships**” box.
- Show **any** “**Edit Relationships**” box (click *B.10.3*) and click on its “**Create New..**” button. You can do this from **any** “**Edit Relationships**” box, **even** if it is a **blank** one. Doing this will show a “**Create New**” dialog box, as the one depicted in the following screenshot:

Select the **master** and **slave** Table-box names by clicking on the **drop-down menu** “**▼**” icon placed on the rightmost side of each of the cells below the labels “**Left Table Name**” and “**Right Table Name**” (respectively). Remind that these drop-down menus will **only** show the names of **unhidden** Table-boxes. **After** you have configured the **master** and **slave** Table-boxes, you configure the first **master-slave field pair** of the Relationship. It is configured by clicking on the **drop-down menu** “**▼**” icon placed on the rightmost side of each of the cells below the labels “**Left Column Name**” (**master** field) and “**Left Column Name**” (**slave** field). Each drop-down menu will **only** show **fields** from the corresponding **master** or **slave Table-box** that you have **previously selected**. Once you are done, click on the “**OK**” button. Doing this will **create** the Relationship and open an “**Edit Relationships**” box.

Be aware that once you click on the “**OK**” button, the configuration you might had performed in the original “**Edit Relationships**” box (where you pressed the “**Create**”

**New..”** button) will be **lost**.

Regardless of how you created the Relationship, you will end up with an “**Edit Relationships**” box, in which the values of the **master** Table-box and one **master** field (left column) plus the **slave** Table-box and one **slave** field (right column) are **already filled-in**. You should use this “**Edit Relationships**” box to **complete** the **configuration** (click *D.9.2*) of your **newly created Relationship**.

I finish this subsection highlighting that it is **very different** to create **one** Relationship linking **two** (or more) field pairs, from creating **two** (or more) Relationships, **each** of them relating **one** field pair.

In case you encountered some problems (but not an **error**), you may click:

- “*D.9.1.2 How do I create a new Relationship in less frequent cases?*”

If you got an **error**, you may click:

- “*L.3 How do I fix errors in my Relationship configuration?*”

### **D.9.1.2 How do I create a new Relationship in less frequent cases?**

This subsection covers the following somehow unusual cases of creating a new Relationship:

- Creating an additional Relationship between two Table-boxes **already having an existing one**
- Creating a Relationship using a **blank “Edit Relationships”** box

#### **Creating an additional Relationship between two Table-boxes already having an existing one**

If you attempt to **create** a **new** Relationship between two Table-boxes that **already have** an **existing** Relationship, MS-Access will show a dialog box asking if you want to **edit** the **already existing** Relationship. If you click “**No**”, a **new** Relationship is created, as you wanted. If you rather click “**Yes**”, you will **view** the **properties** of the **already existing** Relationship (click *B.10.3*), and if you change anything, you will be modifying the **existing** Relationship. You can also cancel the operation by clicking “**Cancel**”.

If you created a **new** Relationship, MS-Access will **add** a **new** Table-box representing the **master** Table and depict the **newly created Relationship** between the **new master** Table-box and the **existing slave** Table-box. MS-Access does this to **avoid having more than one** Relationship between **the same pair** of Table-boxes. If you had **more than one** Relationship between **the same pair** of Table-boxes it would be unclear if it was **one** Relationship with **several** field pairs (each depicted as a connecting line), or **several** Relationships **each** with a **subset of all the field pairs** (each field pair depicted as a connecting line).

#### **Creating a Relationship using a blank “Edit Relationships” box**

If you double-click on the empty background of the “**Relationships**” pane, or on a field name, this will open a **blank “Edit Relationships”** box. You can use this **blank “Edit Relationships”** box to either **edit an existing** Relationship (by modifying the values of its cells), or to **create a new** Relationship **by clicking** on its “**Create New**” button. However, you **cannot** use this **blank “Edit Relationships”** box to create a new

Relationships by directly modifying it: if you attempt to do this, you will be **mistakenly editing** an existing Relationship.

### Getting an error

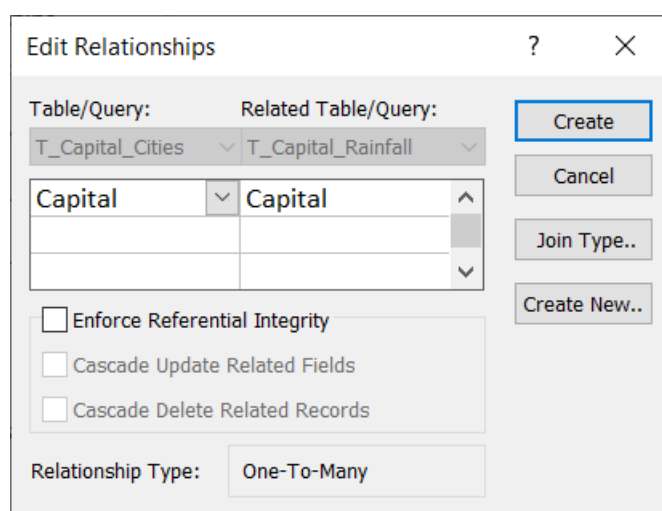
If you got an **error**, you may click:

- “L.3 How do I fix errors in my Relationship configuration?”

## D.9.2 How do I configure a Relationship?

You configure a Relationship with its “**Edit Relationships**” box.

If the “**Edit Relationships**” box is from a **newly created** Relationship (click *D.9.1*), the top-right button will be “**Create**”. If it is rather from an **already existing** Relationship (click *B.10.5*), the top-right button will be “**OK**”. Aside from this difference, the way to configure a **newly created** or an **already existing** Relationship is **exactly the same**. The following screenshot shows an “**Edit Relationships**” box for a **newly created** Relationship (top-right button is “**Create**”):



You can now configure the values of all the cells and checkboxes of the “**Edit Relationship**” box, as I now explain:

- **Column of cells below “Table/Query:” (on the left side)**

This column of cells has at the top cell (right below the label “**Table/Query:**”, with gray background) the **name** of the **master Table-box** in the Relationship. Below the **master** Table-box name it shows cells for the **master fields**. **Each** master field is **related** to the **slave** field placed to its **right**.

The two **master-slave** cells on the **row** right below the **Table-box names** show the **first master-slave** field pair that you indicated when creating this Relationship. You can **link additional master-slave** field pairs in this Relationship by clicking on the rightmost side of each field cell and clicking on the field name you want to relate from the drop-down menu. You can **also** change the first **master-slave** field pair values using the corresponding drop-down menus. If you want to use more field-pairs than can be displayed in the available space, use the scrollbar on the right.

Remind that to have referential integrity (click *D.9.3*), **both fields** in each and every **master-slave** field pair **must** have the **same “Field Type”** and also the **same “Field Size”**. Additionally, the **master field(s)** **must** have an associated **index without**

**duplicate values** and **without Nulls**.

Notice that you **must** select one field name on the “**Table/Query:**” column for each selected field on the “**Related Table/Query:**” column (and vice versa): MS-Access will **not allow** you to select a field name in **only one** of the columns, because it is meaningless to have a field related with nothing.

- “**Related Table/Query:**” column of boxes (on the right side)

This column of cells shows at the top cell (right below the label “**Related Table/Query:**”, with gray background) the **name** of the **slave Table-box** in the Relationship. Below the slave Table-box name it shows cells for the **slave fields**. **Each** slave field is **related** to the **master** field placed to its **left**.

The two **master-slave** cells on the **row** right below the **Table-box names** show the **first** master-slave field pair that you indicated when creating this Relationship. . You can **link additional master-slave** field pairs in this Relationship by clicking on the rightmost side of each field cell and clicking on the field name you want to relate from the drop-down menu. You can **also** change the first **master-slave** field pair values using the corresponding drop-down menus. If you want to use more field-pairs than can be displayed in the available space, use the scrollbar on the right.

Remind that to have referential integrity (click *D.9.3*), **both fields** in each and every **master-slave** field pair **must** have the **same “Field Type”** and also the **same “Field Size”**. Additionally, the **master field(s)** **must** have an associated **index without duplicate values** and **without Nulls**.

Notice that you **must** select one field name on the “**Table/Query:**” column for each selected field on the “**Related Table/Query:**” column (and vice versa): MS-Access will **not allow** you to select a field name in **only one** of the columns, because it is meaningless to have a field related with nothing.

- “**Enforce Referential Integrity**” checkbox

If you **tick** this checkbox, the system will **enforce referential integrity** (click “*C.11.1 What is a Relationship with referential integrity?*”) in this Relationship.

I strongly advise you **always** tick “**Enforce Referential Integrity**”: otherwise the Relationship’s usefulness is extremely limited (“*D.9.3 What is the effect of “Enforce Referential Integrity”?*”). Remind that to have referential integrity, both fields in **each and every pair of master-slave fields** **must** be of the **same field type** and also of the **same field size**. Additionally, the **master field(s)** must have an associated **index without duplicate values** and **without Nulls**.

- “**Cascade Update Related Fields**” checkbox

If you **tick** this checkbox, the system will **automatically update** the **values** of **all** corresponding **slave field(s)** in **all slave records** when you **change the value** of the **master field(s)** in a given **master record**. This means, for example, that if a street name is changed in the **master** Table of street names, the new street name will appear automatically in the corresponding field of all records linked to this one throughout the database. As you may see, this is extremely convenient. Notice you can only check this checkbox if you have already set “**Enforce Referential Integrity**”.

I **strongly** recommend you **always** set “**Cascade Update Related Fields**”, unless you

have a very sound reason not to do it.

If you want to know more about this, you may click “[D.9.4 What is the effect of “Cascade Update Related Fields”?](#)”.

- **“Cascade Delete Related Records” checkbox**

If you **tick** this checkbox, the system will **automatically delete all** orphaned records. This means, for example, that if a house record is removed from the addresses **master** Table, the system will **automatically delete all the records** that were linked to this address from **all the Tables** in the database.

If you **do not tick** this checkbox, the system **will not allow you to remove** a record until you have **previously removed all** the records that point to it.

**Tick**ing this checkbox may seem convenient, but **it is extremely risky**. I strongly advise you **never** set “**Cascade Delete Related Records**”. It may take a little more manual work if you want to remove a record from a **master** Table, but it is **much safer** than losing your valuable data (i.e., the orphaned records) by mistake.

If you want to know more about this, you may click “[D.9.5 What is the effect of “Cascade Delete Related Records”?](#)”.

- **“Relationship Type” cell**

This cell **shows** the type of this Relationship. The Relationship type can be **one-to-many**, **one-to-one** or **indeterminate**. MS-Access **fills-in** this cell with the correct Relationship type, based on the characteristics of the fields you have linked in the “**Table/Query**” and “**Related Table/Query**” boxes. The value of this cell is defined as follows:

- If the **master** field(s) **are** a candidate **Key**, the **slave** fields **are not** a candidate **Key**, and the two field types of **every master-slave** field pair are the same, then the Relationship is **one-to-many**.
- If the **master** field(s) **are** a candidate **Key**, the slave fields **are also** a candidate **Key**, and the two field types of **every master-slave** field pair are the same, then the Relationship is **one-to-one**.
- In any other case, the Relationship is **indeterminate**.

Notice that MS-Access will change the value of the “**Relationship Type**” cell as you change the fields you are linking in the “**Table/Query**” and “**Related Table/Query**” cells (see at the top of this bullet list).

- **“Create” or “OK” button**

This button closes the “**Edit Relationships**” box, saving changes. If the “**Edit Relationships**” box was of a **new** Relationship, the button will show “**Create**”: if you **click it** the Relationship will be **created**. If the “**Edit Relationships**” box was of an **existing** Relationship, the button will show “**OK**”: if you **click it** the **changes** you did to the Relationship will be **saved**.

- **“Cancel” button**

This button closes the “**Edit Relationships**” box, **discarding** all the **changes** that you did.

- **“Join Type..” button**

This button allows to select the default join type in Queries defined with the graphical interface of MS-Access. My advice is you **ignore** this button.

- **“Create New..” button**

Clicking it shows the **“Create New”** box to create a new Relationship (click *D.9.1.1*). Notice that once you click the **“OK”** button in the **“Create New”** box, the **configuration** you did in the **“Edit Relationships”** box (where you just clicked the **“Create New..”** button) will be **lost**.

Remind that when you check both **“Enforce Referential Integrity”** and **“Cascade Update Related Fields”**, the **values** of the **slave fields** in **each record** in the **slave Table** will correspond to the values of the **corresponding master fields** in its corresponding **record** from the **master Table**. To say it in other words, the linked fields from the slave Table **cannot** have **their own values**, and rather, can **only** have the values of **one record** from the **master Table**.

Once you are done with the configuration of your new Relationship, click on the **“Create”** or **“OK”** button and the Relationship has been configured.

If you got an **error**, you may click:

- *“L.3 How do I fix errors in my Relationship configuration?”*

### **D.9.3 What is the effect of “Enforce Referential Integrity”?**

Setting **“Enforce Referential Integrity”** in a Relationship makes MS-Access **automatically guarantee** that the slave **field value requirements** that define the Relationship **are always satisfied**.

I **definitively** advise you to **always** tick **“Enforce Referential Integrity”** in every Relationship that you create. Recall from *“C.11.1 What is a Relationship with referential integrity?”* that in order to have Referential Integrity, the Relationship must fulfil two conditions:

1. **Both fields** of each **master-slave field pair** have the **same field type** and the **same field size**.
2. The **master field(s)** has/have an **index without duplicate values** and **without Nulls**.

The second condition is equivalent to say that the **master fields** are a candidate Key (click *C.10.3*) of the Table. Remind that if the **master fields** are the Table’s **Primary Key field(s)**, they **always** have an **index without duplicate values and without Nulls**.

If, **and only if**, you tick **“Enforce Referential Integrity”** in a Relationship, MS-Access will allow you to **also** tick **“Cascade Update Related Fields”** and/or tick **“Cascade Update Related Fields”**.

### **D.9.4 What is the effect of “Cascade Update Related Fields”?**

Tickling **“Cascade Update Related Fields”** makes MS-Access **automatically update** the value of the **slave field(s)**, in **all the slave records**, to the **new value** of the **master field(s)** in **their** corresponding **master record**, each time that the **value** of the **master field(s)** has been modified in a given **master record**.

If in a Relationship you tick “**Enforce Referential Integrity**”, but you **do not set “Cascade Update Related Fields”**, MS-Access will **not** automatically update the **change of value** of a **master field** from a master record on its **slave field(s)** from **all its slave records**. You can then have inconsistent data in respect to the Relationships that you configured.

I strongly advise you **always tick “Cascade Update Related Fields”**.

Notice that when you change the value of a **master field**, this will automatically update the values of **all** its **slave fields** in **all** its slave record, and each updated **slave record** will be subject to field error checks and record error checks (click *L.4.3.4*).

### **D.9.5 What is the effect of “Cascade Delete Related Records”?**

Ticking “**Cascade Delete Related Records**” makes MS-Access **automatically delete all slave records** when you delete **their master record**.

If in a Relationship you set “**Enforce Referential Integrity**” but you **do not set “Cascade Delete Related Records”**, MS-Access **will not allow** you to **delete any master record** that has one or more **slave records**. If you **try** to delete **such a master record**, MS-Access will **not** do it, and will show an informative message indicating that there are slave records dependent on this one.

Setting “**Cascade Delete Related Records**” may seem convenient, but it is **extremely risky**. I strongly advise you **do not set “Cascade Delete Related Records”**.

### **D.9.6 What Relationships should I configure?**

You configure a **one-to-many** Relationship when you have one (or more) **master Tables** in your database, **each** of which lists “**all the**” **entities** of a given class that **exist** in this database. The class of entities in **each** of the **master Tables** could be for example **countries, persons, products, cities, cars, addresses**, etc. You also have in your database **other Tables** where you want to include in their fields **references** to **countries, persons, products**, etc. Then, you should establish several **one-to-many** Relationships, each of them from **a given master Table**, to **each of the other slave Tables** that include in their fields a **reference** to the entities listed in their **corresponding master Table**. The **one-to-many** Relationship is the **most frequent** type of Relationship.

The **one-to-one** Relationship is used typically when you have an entity class (e.g., **project, order**, etc.) that has a number of fields that only apply to **some entity cases**, and you do not want the main entity Table with **so many optional** fields. Then you have the main entity Table (the **master Table**) with the mandatory fields that apply to **every** entity, and also, you list the optional type-specific fields in one (or more) **slave Tables**. In this case, you establish several **one-to-one** Relationships: each Relationship is established from the **master Table** to **each** of the **slave Tables**.

A **many-to-many** “*Relationship*” is created when you need to record in your database how two (or more) entities relate among themselves. Then you create a **many-to-many** Relationship between the master Tables of the entities you want to relate among themselves.

When creating **one-to-one** or **one-to-many** Relationships or **many-to-many** “*Relationships*”, recall my advice to **always configure “Enforce Referential Integrity”** and “**Cascade Update Related Fields**” options, and **never** configure “**Cascade Delete**

**Related Records**". The two first options are very useful and safe, while the third one may be useful, but it is **extremely risky**.

If you want to know more about the concepts behind Relationships, you may click "*C.11 What is a Relationship?*".

## **D.10 How do I design MS-Access Forms?**

You may click:

- "*D.10.1 What are MS-Access Forms?*"
- "*D.10.2 How do I create an MS-Access Form?*"
- "*D.10.3 How do I add/delete/update a field of a Form?*"
- "*D.10.4 How do I configure VBA action code associated to Form Events?*"

### **D.10.1 What are MS-Access Forms?**




MS-Access Forms are objects that allow to **enter** data in very flexible way into **one or more** Tables. Forms can also be used to **present** data from one or more Tables and Queries, but this can also be done by Queries. Forms therefore are very useful for **entering** data. Forms are very powerful and can be configured to perform different actions (assign values, invoke user-defined VBA subroutines, ...) upon different Form events (field update, record update, mouse over a field, Form closed, ...).

In its simplest shape, a **Form is like a cover you put over one Table**, where you can automatically do more things. The Table where the Form's data is taken from is called its record-source Table. Some examples of things you can do with Forms are:

- Calculate the values of one, or more, **auxiliary** Tables every time that you modify the data of the Table associated to the Form.
- Compute the default value of a Table field, depending on the values of other fields in the same record and using **user-defined functions**.
- Compute the default value of a field using **auxiliary Queries**.

In a more complex shape, a Form may take data from several related Tables and from your database Queries.

### **D.10.2 How do I create an MS-Access Form?**



To create a Form over one Table, click on "**Create**" from the "Ribbon-bar", and then on the **Form Wizard**  icon, placed in the top right corner of the "**Forms**" Ribbon group. In the "**Form Wizard**" click on the **drop-down menu**  icon and click on the Table you want the Form to be based on. Then click on the **add fields**  icon and all the Table's field will be selected into the Form. Click on "**Next**" and click on "**Datasheet**". You can now type-in the name you want for this Form in the text box at the top of the Wizard window. Then click on "**Finish**" and the form will be created.

Each Form is placed as an object in the "**Navigation Pane**" (click *B.4*), under the collection "**Forms**" with its corresponding name. You can open, copy, paste, rename or delete a Form, just like you can do with any other object in the "**Navigation Pane**".

When a Form is created from a Table, the form inherits all the configuration of the



Table. All the formatting, and all the drop-down menus of the Table are inherited by the Form. However, once the Form has been created, the Form and the Table can be independently modified of each other, and each will keep its own configuration. For example, if you add or modify a field drop-down menu in the Table, this change **will not** be automatically propagated to the Form(s) based on that Table.

To create a Form over several Tables/Queries, click on the “**Create**” Ribbon, and then on the **Form Wizard**  icon, placed in the top right corner of the “**Forms**” Ribbon group. In the “**Form Wizard**” dialog box click on the **drop-down menu**  icon and click on the first Table/Query you want the Form to be based on. Then double-click on each field from the Table/Query that you want to incorporate to the Form. When you are done with the fields of this Table/Query, you **select** another Table/Query using the drop-down menu at the top of the Wizard. You do this for as many Table/Queries you want. When you are done incorporating fields from your Table/Queries, click on “**Next**”. You then click on “**Datasheet**”. You can now type-in the name you want for this Form in the text box at the top of the Wizard window. Then click on “**Finish**” and the Form will be created.


### **D.10.3 How do I add/delete/update a field of a Form?**

You may click:

- “*D.10.3.1 How do I add a field to a Form?*”
- “*D.10.3.2 How do I delete a field from a Form?*”
- “*D.10.3.3 How do I update a field of a Form?*”

#### **D.10.3.1 How do I add a field to a Form?**

Open the Form in “**Datasheet View**” (click *B.4.1.3*).

Click on “**Datasheet**” from the “Ribbon-bar” (below the “**Form Tools**” contextual label). Then click on the **Add Existing Fields**  icon from the “**Tools**” Ribbon group. This will open the “**Field List**” sub-pane at the right side of the “Form pane”. There you will see the fields of the associated Table. To add a field, just drag-and-drop it. This is, press the left mouse button over the field name from the “**Field List**” sub-pane, and **without releasing** the mouse button, you move the mouse to the place in the Form where you want this field inserted. MS-Access will you a **vertical pink line** indicating where the field will be inserted, that will change place when you change the mouse location. Once you see the pink line where you want the field inserted, you release the mouse button.

Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form design. You may then use the Form in “**Datasheet View**” or **close** the Form (click *B.4.1.7*).

#### **D.10.3.2 How do I delete a field from a Form?**

Open the Form in “**Datasheet View**” (click *B.4.1.3*).

You can **delete** a Form field in either of the following ways:

- **Right-click** on the field **header** and click on “**Delete**” from the pop-up menu.
- **Select** a field, or a range of fields (click *B.6.1.4*). You then either:

- Press the “**Supr**” key.
- **Right-click** on the field header of any of the selected fields, and click on “**Delete**” from the pop-up menu.

Regardless of the way you do it, you will get a confirmation window. If you confirm the deletion, the Form field(s) will be **deleted**.

Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form design. You may then use the Form in “**Datasheet View**” or **close** the Form (click *B.4.1.7*).

### **D.10.3.3 How do I update a field of a Form?**

Open the Form in “**Datasheet View**” (click *B.4.1.3*).

If you want to **update the configuration** of a Form field in order to match the current configuration of its associated Table field (e.g., a drop-down menu that you configured in the Table), you just **delete** the field (click *D.10.3.2*) and then **add** it back (click *D.10.3.1*). Notice that all the configuration you had possibly done on this Form’s field **will be lost** and the field will have the configuration inherited from the corresponding Table field that you have just added.

Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form design. You may then use the Form in “**Datasheet View**” or **close** the Form (click *B.4.1.7*).

### **D.10.4 How do I configure VBA action code associated to Form Events?**

If you had not previously done this, you configure this Form to have an associated VBA Module. If you want to know how to do this, you may click:

- “*D.10.4.1 How do I configure a Form to have an associated VBA Module?*”


You then click on the subsection you want:

- “*D.10.4.2 How do I enter/edit the VBA subroutine associated to a Form Event?*”
- “*D.10.4.3 How does my VBA code modify field values in the Form record being edited?*”
- “*D.10.4.4 Why should I write options in the VBA Modules of my Forms?*”
- “*D.10.4.5 How do I close the VBA editor?*”

#### **D.10.4.1 How do I configure a Form to have an associated VBA Module?**

Open the Form in “**Design View**” (click *B.4.1.3*).

Show the Form’s “**Property Sheet**” (click *B.8.1*).

You now click on the **drop-down menu** “” icon of the “**Property Sheet**” and click on “**Form**” from the drop-down menu. Then click on the “**Other**” tab, locate the property “**Has Module**” and on the cell to the right of “**Has Module**” you configure “**Yes**” (either typing-in or using the drop-down menu). Notice you **cannot** configure this property from “**Datasheet View**” and you **must** do it from “**Design View**”.

You only need to do this **once** for each Form. After it is configured, the Form will keep its associated VBA Module and you can change its VBA code as I will show right now.


Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form design. You may then **close** the Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

#### **D.10.4.2 How do I enter/edit the VBA subroutine associated to a Form Event?**

If not already done, configure the Form to have an associated VBA Module (click *D.10.4.1*).


Open the Form in “**Datasheet View**” (click *B.4.1.3*).

Show the Form’s “**Property Sheet**” (click *B.8.1*).

You now click on the **drop-down menu** “” icon and click on the Form **element** to whose **Events** you want to associate some VBA code. The Form elements you can select from the drop-down menu to configure its **Events** are the following:

- **Form**: the **Form** as a whole.
- “**Column\_name**”: the **column** (field) of the Form (the one named “**Column\_name**”).
- **FormFooter**: the **footer section** of the Form.
- **FormHeader**: the **header section** of the Form.
- **Detail**: the **detail section** of the Form.
- The **Label** elements **do not have** any associated **Event**.

Now click on the “**Events**” tab, and you will see all the **Events** (“**On Click**”, “**Before Update**”, “**After Update**”, “**On Got Focus**”, “**On Dbl Click**” and many others), one in each row, that are associated to the Form **element** you selected. If an **Event** already has an VBA Subroutine associated to it, you will see the text “[**Event Procedure**]” on the cell to the right of the **Event** name.

To enter/edit the VBA subroutine associated to one of these **Events**, click on the rightmost end of the **Event** row. In case the row was selected, you should click on the icon with three dots “” that you see on its rightmost end. If this **Event** already had some associated VBA code, the **VBA editor window** will open, showing the VBA code of the **Event**. If this **Event** did not have yet any associated VBA code, a dialogue-box will be shown, where you should double-click on “**Code Builder**”, and this will open the **VBA editor window**.

The VBA code associated to each **Event** is enclosed in a VBA private subroutine (i.e., a procedure, which is similar to a function with no arguments) written in the VBA module associated to the Form. The name of the VBA subroutine is the name of the element followed by underscore and the name of the **Event**. For example, the subroutine associated to the **Event** “**Close**” in the element “**Form**” is named “**Form\_Close()**”, the one of the **Event** “**AfterUpdate**” in a column named “**CalYear**” is named “**CalYear\_AfterUpdate()**”, and the one of the **Event** “**Click**” in the Form header is named “**FormHeader\_Click()**”.

**All** the VBA **Event** subroutines of the Form will be in the VBA Module associated to

the Form. Therefore, when the VBA editor window is shown, you will see **not only** the subroutine you just clicked (the one associated to the **Event** where you clicked on its rightmost side), but actually **you will see all the VBA subroutines** that you have defined up to now in this Form. Not only you **see** all the subroutines, but you can **edit all the ones** that you want and/or **enter** any number of **new** subroutines that you want.

I now show you two very simple examples of **Event** subroutines. The first one is the subroutine associated to the **Event “Close”** in the element **“Form”**:

```
Private Sub Form_Close()
    Dim Ignore_result As String
    Ignore_result = Generate_Auxiliary_Table_for_Cities()
End Sub
```

This subroutine is invoked by MS-Access every time that you **close** the Form. As you may see, the subroutine invokes the function:

**“Generate\_Auxiliary\_Table\_for\_Cities()”**

which could for example update the content of an auxiliary Table, that depends on the Table associated to this Form.

The second example is the subroutine associated to the **Event “AfterUpdate”** of a Form column named **“CalYear”**.

```
Private Sub CalYear_AfterUpdate()
    Me.First_day_of_year = DateSerial(Me.CalYear, 1, 1)
    Me.Last_day_of_year = DateSerial(Me.CalYear, 12, 31)
End Sub
```

This subroutine is invoked by MS-Access every time that you update the value of a cell/field in the column/field name **“CalYear”**. As you may see, the subroutine automatically introduces the date of the first day, and the last day, of the year **“CalYear”** in the fields **“First\_day\_of\_year”** and **“Last\_day\_of\_year”** of this record.

In case you want to know more about how to modify the values of the fields of the Form record being edited, you may check the next section *D.10.4.3*.

If you want to know more about how to write VBA code, you may click *“K.9 How do I write my user-defined VBA functions and database Subroutines?”*.

Once you are done writing your VBA code, you should save/close the VBA editor, as I explain in *“D.10.4.5 How do I close the VBA editor?”*.

### **D.10.4.3 How does my VBA code modify field values in the Form record being edited?**

One quite frequent application of Forms is to provide **default values** that depend on the **values of other fields** and/or that are **computed from Queries**. Notice that neither of these functions can be achieved with a plain Table.

In order to enter **values** into a Form **record**, MS-Access provides the built-in **“Me”** data structure. The **“Me”** VBA data structure has one field for each of the fields in the Form. The names of the fields of the **“Me”** structure are the same ones as the Form fields. The values of the **“Me”** structure fields are the same as the ones of the fields of the **record being edited** (regardless of it being a **new** record or an **existing** record).

Therefore, the **“Me”** built-in structure allows that in your VBA code you can **access** the

**current values** of the fields of the record being edited, and also, that you **can modify the values** of the fields of the record being edited. To show an example, let me bring back the Table of quarterly rainfall measurements in capital cities you created in A.5:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

In this Table, you could create a function to enter a default value for the field “Quart\_Rainfall” that depends on the value of the field “Capital”. The function could look like:

```
Private Sub Capital_AfterUpdate()
    If Me.Capital = "Washington"
        Then Me.Quart_Rainfall = 4
        Else Me.Quart_Rainfall = 10
    End If
End Sub
```

This subroutine will be invoked each time you edit the value of field “Capital”. The effect of invoking it will be to enter the value “4” in field “Quart\_Rainfall” if the value of “Capital” is “Washington” and will enter the value “10” otherwise.

#### **D.10.4.4 Why should I write options in the VBA Modules of my Forms?**

Because it you will achieve a faster code and a coherent handling of text strings.

I advise you write the following two options at the top of each VBA Module associated to a Form:

```
Option Compare Database
Option Explicit
```

The option “**Compare Database**” will make string comparison be based on the same ordering of your MS-Access database. This is extremely useful to avoid puzzling results arising from handling **two different** ordering criteria for *Strings*.

The option “**Explicit**” obliges you to write a variable declaration of all the variables you use in your VBA code. This may sound like more work, but it is **much, much, safer** and it is certainly worth the extra work. It has an added advantage that your VBA code will **run faster**.

Below these two options you may write one after another all your VBA **Event** subroutines.

#### **D.10.4.5 How do I close the VBA editor?**

When you are done writing your VBA **Event** subroutines, I advise you to compile your

VBA module by clicking on “**Debug**” (placed at the top of the VBA editor window) and then clicking on “**Compile Database1**” from the pop-up menu. If you get compilation errors, debug your code.

Once the compilation returns no errors, close the VBA editor window by clicking on the close icon “**X**” on its top-right corner. Notice that closing the VBA editor will **automatically** save all your VBA code changes.

## **D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?**

I explain in this chapter the **procedure** to set up and configure different ways of entering data into your Table/form fields. However, I do not provide in this chapter **advice** on how to **make the most** of drop-down menus. If you want to know more about good practices on drop-down menu configuration, you may click “*K.1.8 What are good practices in configuring my drop-down menus?*”.

You can configure the following ways to enter data in a Table/Form field:

- **Typing-in** the values  
Set “**Display Control**” as “**Text Box**”: see further below.
- **Selecting** values from a **drop-down menu**  
Set “**Display Control**” as “**Combo Box**” (or alternatively as “**List Box**”, but I do not recommend it): see further below.
- **Selecting** values with a **checkbox** (only for **Yes/No** fields)  
Set “**Display Control**” as “**Check Box**”: see further below.
- **Selecting** values with a **date-picker** (only for **Date/Time** fields)  
See further below.

The specific **data entering mechanism(s)** that you can configure for **each** Table/Form field depend(s) on its **field type**, as follows:

- **Number, Large Number** or **Short Text** fields  
**Can have type-in** (default) **and** a **drop-down menu**. You may click:
  - “*D.11.1 How do I configure a drop-down menu to enter data in a Table field?*”
  - “*D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?*”
  - “*D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?*”
- **Date/Time** fields  
**Can have type-in** (default) **and** a **date-picker**, but **not** a drop-down menu (nor “**Lookup**” options). You may click:
  - “*D.11.2 How do I configure a date-picker to enter a date in a Table field?*”
  - “*D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?*”

Even though MS-Access does not provide a direct way to configure a drop-down menu in a **Date/Time** field, if you really want to do it, there is a **trick** that I describe in “*K.1.9 How do I configure a drop-down menu in a Date/Time field?*”.

- **Yes/No** fields  
**Can have checkbox** (default) **or type-in**. They can **also** have a drop-down menu, but I consider it is better to use a **checkbox** instead. You may click:
  - *“D.11.3 How do I configure type-in or checkbox to enter data in a Table/Form field?”*
- **Long Text, Currency, Calculated, AutoNumber, OLE Object, Hyperlink or Attachment** fields  
**Can only have type-in** (cannot have a drop-down menu, nor **“Lookup”** options). There is one exception listed in the next bullet point.
- **Calculated** fields with **“Result Type = Yes/No”**  
**Can have a checkbox** (as an alternative to the default **“Text Box”**). Since the field value is **calculated** (i.e., it is **not** entered), the **“Check Box”** in this case is just a way to **show** the value, and **not** a way to **enter** it.

Notice that you **do not enter** values in **AutoNumber** fields, and therefore, they do not have any configuration for entering data into them (their **“Lookup”** tab is blank).

Notice from the sections referenced above that drop-down menus and date-pickers are configured **differently** in **Tables** and **Forms**.

If you configured a drop-down menu, and you want to go back to type-in only, you may click:

- *“D.11.3 How do I configure type-in or checkbox to enter data in a Table/Form field?”*

### **D.11.1 How do I configure a drop-down menu to enter data in a Table field?**

You can **only** configure a drop-down menu in a field with **field type Number, Large Number or Short Text**. Notice **Yes/No** fields can **also** have a drop-down menu, but I consider it is better to use a **checkbox** instead.

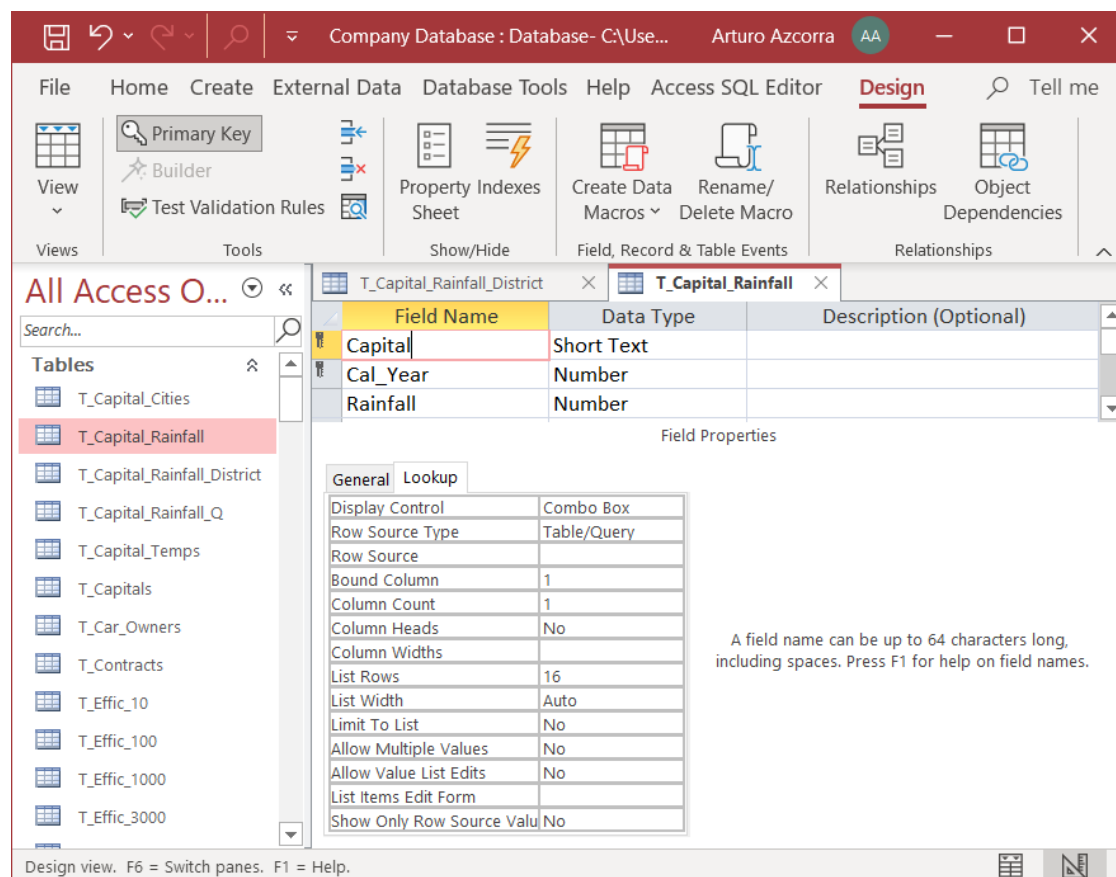
Open the Table in **“Design View”** (click *B.4.1.3*). Click on the field row of the field that you want to have a drop-down menu, and then click on the **“Lookup” tab** in the bottom sub-pane.

To configure a drop-down menu, you have to configure the **“Lookup”** properties: **“Display Control”**, **“Row Source Type”** and **“Row Source”** as follows.

Click on the rightmost side of the first row (named **“Display Control”**) of the bottom sub-pane and click on **“Combo Box”** from the drop-down menu. Notice that clicking on **“List Box”** will also configure a drop-down menu, but with **less** configuration options than **“Combo Box”**. My advice is therefore you always use **“Combo Box”** for your drop-down menus.

In case you decide to select **“List Box”**, you should be aware that in a **Table** it will **allow** you to input a value other than the ones in the list (e.g., typing it in, or editing the value selected from the list), but in a **Form** it **will not** allow you to input a value other than the ones in the list.

If you configured “**Combo Box**”, this will show a number of property rows in the bottom “**Lookup**” sub-pane, below the first row named “**Display Control**”. These property rows are shown in the following screenshot:



Click on the rightmost side of the property row labeled “**Row Source Type**” and the drop-down menu will show you the following three options (for each of them I indicate the corresponding explanatory subsection):

- “**Table/Query**”  
Click “[D.11.1.1 How do I configure a Table drop-down menu that takes its values from a Table/Query?](#)”.
- “**Value List**”  
Click “[D.11.1.2 How do I configure a Table drop-down menu that takes its values from a Value List?](#)”.
- “**Field List**”  
Click “[D.11.1.3 How do I configure a Table drop-down menu that takes its values from a Field List?](#)”.

Configuring drop-down menus in Table’s fields **does not** affect **stored data** and has **no side effects**.

#### **D.11.1.1 How do I configure a Table drop-down menu that takes its values from a Table/Query?**

If you set “**Row Source Type**” as “**Table/Query**”, the values shown in the drop-down menu are the values of the **first field** of all the records of a **Table name**, **Query name**,



or **SQL operation**. The specific **Table name**, **Query name**, or **SQL operation** is the one that you type-in in the property row “**Row Source**”. The SQL operation can be a **Select** operation, a **Union** operation or a **Transform** operation.

If you write a **Table name** as “**Row Source**”, the **order of values** in the drop-down menu will be the one of the first field of that Table. If the first field is indexed, it will be the order of the index that you configured. If the first field is not indexed, it will be the order of records as they were introduced in that Table. The ordering options that you may configure for the “**Row Source**” Table in its “**Datasheet View**” **does not** affect the order of values in the menu.

If you write a **Query name** or **SQL operation** as “**Row Source**”, the **order of values** in the drop-down menu will be the one of the output record-list of the Query or SQL operation.

A quite frequent case is writing a **Select** operation in the “**Row Source**” property. The great advantages of using a **Select** operation are:

- You can **select the field name you want** from all the ones in the Table name or Query name, and you are therefore **not restricted** to using only the first field name.
- You can use the “**ORDER BY**” clause to define the **exact order you want** for the values in the drop-down menu (click *F.7.12*).
- You can use the “**DISTINCT**” clause (click *F.7.11*) to suppress duplicate values, in case they exist. Duplicate values are **in principle** undesirable in a drop-down menu.

An extremely useful approach is to produce drop-down menus with a **Select operation** over an **auxiliary Table**. The auxiliary Table allows you to modify the values when you want, and they will be automatically updated for **all** the drop-down menus over that Table. The auxiliary Table may have, **in addition** to the **field** with the drop-down menu **values**, a field with a **descriptive text**, a field with **comments**, a field to **order the values** over it, or any **other auxiliary** fields that you may need.

If you want to configure different options in a “**Combo Box**”, you may click:

- “*D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?”*”

If you want to know some good practices in configuring your drop-down menu, you may click:

- “*K.1.8 What are good practices in configuring my drop-down menus?”*”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?”*”.

#### **D.11.1.2 How do I configure a Table drop-down menu that takes its values from a Value List?**

If you set “**Row Source Type**” as “**Value List**”, the values shown in the drop-down menu are a **list of values you type-in** in the property row “**Row Source**”.

The **order of values** in the drop-down menu will be the same as the order of values that **you have typed-in** into “**Row Source**”.

You have to separate the values in the list with a semicolon “;” character. If the values are text strings, enclose each text string in double quotes. A couple examples of value lists that can be typed-in into “**Row Source**” are:

- **Number** value list: **0 ; 0.5 ; 1 ; 1.5**
- **Short Text** value list: **"Q1" ; "Q2" ; "Q3" ; "Q4"**

Use a “**Value List**” when the list of values is reasonably short and you do not expect the list to change along the lifetime of the database (e.g., to enter identifiers for quarters). If the value list is long (e.g., 40 values) or it will change along the database lifetime (product identifiers, that come and go), it is better to use the “**Table/Query**” value as “**Row Source Type**”.

If you want to configure different options in a “**Combo Box**”, you may click:

- “*D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?*”

If you want to know some good practices in configuring your drop-down menu, you may click:

- “*K.1.8 What are good practices in configuring my drop-down menus?*”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

### **D.11.1.3 How do I configure a Table drop-down menu that takes its values from a Field List?**

If you set “**Row Source Type**” as “**Field List**”, the values shown in the drop-down menu are the **field names** of a **Table name** or **Query name**. The specific **Table name**, or **Query name** is the one that you type-in in the property row “**Row Source**”.

The **order of values** in the drop-down menu will be the same as the **field order** in the Table (in “**Design View**”) or in the Query (in “**SQL View**”).

If you click on an option from the drop-down menu, its value will be entered in the field as if it were typed-in. This means that if the field type is **Short Text**, the value will be a text string. If the field type is **Number** or **Large Number** and the option from the menu (i.e., the field name) looks like a number value (e.g., 1 or 24), it will be correctly entered into the field.

Using “**Field List**” as “**Row Source Type**” is particularly useful when using a **Transform Query** as the “**Row Source**”, because the field names of a **Transform Query** are actually **values** that have been turned into field names.

If you want to configure different options in a “**Combo Box**”, you may click:

- “*D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?*”

If you want to know some good practices in configuring your drop-down menu, you may click:

- “*K.1.8 What are good practices in configuring my drop-down menus?*”

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table

design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

### **D.11.2 How do I configure a date-picker to enter a date in a Table field?**

You can **only** configure a date-picker in a field with **field type Date/Time**.

Right-click on the Table name in the “**Navigation Pane**” and click on “**Design View**” from the pop-up menu.

Click on the **row** of the field you want to have a date-picker. In the “**General**” tab (in the bottom sub-pane) locate the row named “**Show Date Picker**”. Click on the rightmost side of that row and click on “**For dates**” from the drop-down menu.

Once you are done with your Table configuration, **save** (click *B.4.1.6*) your Table design. You may then **close** the Table (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

I do not find myself a date-picker very useful, and on my view it is better to configure a default **Date/Time** value (e.g., with the function “**Date()**” to insert today’s date with zero-time), that you can edit in the field to change it to some other value you want.

Although MS-Access **does not** allow to configure a drop-down menu in a **Date/Time** field, if you really need it, there is a trick that I explain in “*K.1.9 How do I configure a drop-down menu in a Date/Time field?*”.

### **D.11.3 How do I configure type-in or checkbox to enter data in a Table/Form field?**

This depends on the **field type**, as follows.

#### **For all field types except Yes/No and Date/Time**

For these field types, **type-in** is the **default** way to enter data. Even if you configured a drop-down menu (on some field types only), you can **still** type-in your values.

If what you want is to have **only** type-in (i.e., **remove** the drop-down menu), you have to follow the same procedure as when setting the drop-down menu, but setting “**Display Control**” as “**Text Box**”. To do this, you may click:

- “*D.11.1 How do I configure a drop-down menu to enter data in a Table field?*”
- “*D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?*”

#### **For a Yes/No field type**

For a **Yes/No** field type, **checkbox** is the **default** way to enter data. If you want to change it to **type-in**, you have to follow the same procedure as when setting a drop-down menu, but setting “**Display Control**” as “**Text Box**”. To do this, you may click:

- “*D.11.1 How do I configure a drop-down menu to enter data in a Table field?*”
- “*D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?*”

### For a Date/Time field type

For a **Date/Time** field type, **type-in** is the **default** way to enter data. Even if you configured a date-picker, you can **still** type-in your values.

If what you want is to have **only** type-in (i.e., **remove** the date-picker), you have to follow the same procedure as when setting it, but clearing the corresponding option. To do this, you may click:

- “*D.11.2 How do I configure a date-picker to enter a date in a Table field?”*”
- “*D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?”*”

### D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?

When creating a Form from a record-source Table, each and every Form field will inherit the drop-down menu or the date-picker of its corresponding record-source Table field. However, if you **later** do any **change** (create, delete, modify) on a field drop-down menu in a record-source Table of the Form, the **change** you did **will not propagate** automatically to the Form. If you want the change you did to propagate to the Form, you can either manually introduce the same change in the Form or delete the Form and create it again from the record-source Table, so it inherits all its drop-down menus and date-pickers. Remind my advice of using a default value for dates (e.g., today’s date with function “**Date()**”) instead of using a date-picker.

A **drop-down** menu is configured **differently** in Forms depending on whether it is an **existing drop-down** menu or a **new** one. You may click on:

- “*D.11.4.1 How do I configure a new drop-down menu directly in a Form field?”*”
- “*D.11.4.2 How do I configure an existing drop-down menu in a Form field?”*”
- “*D.11.4.3 How do I configure a new date-picker directly in a Form field?”*”

#### D.11.4.1 How do I configure a new drop-down menu directly in a Form field?

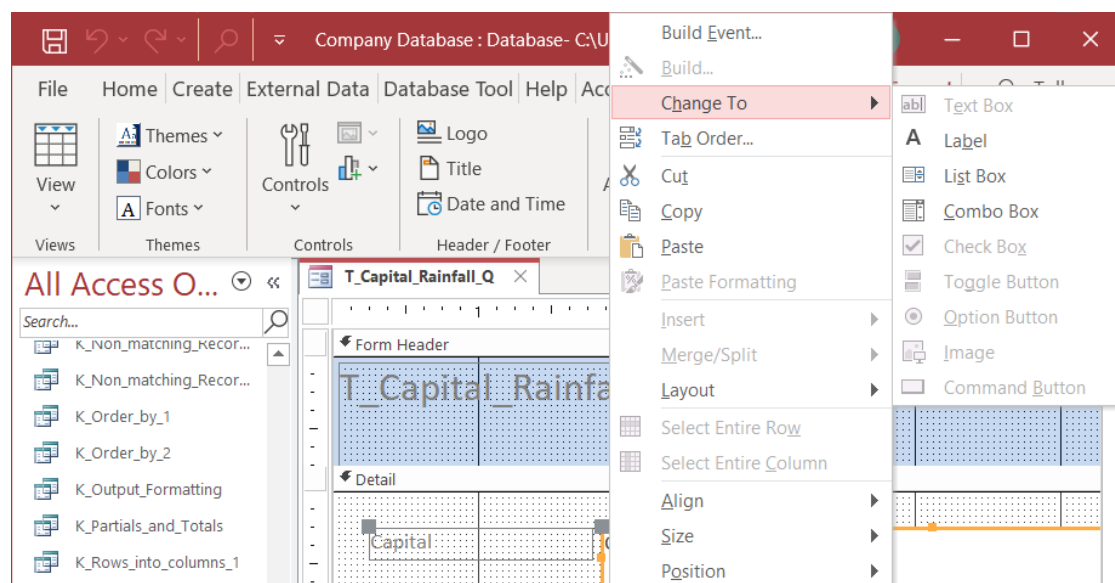
You can **only** configure a drop-down menu in a field with **field type Number, Large Number** or **Short Text**. Notice that **Yes/No** fields can **also** have a drop-down menu, but I consider it is better to use a **checkbox** instead.

If what you want is to **change** the configuration of an **existing** drop-down menu in a Form field, you should read “*D.11.4.2 How do I configure an existing drop-down menu in a Form field?”*”.

Otherwise, open the Form in “**Design View** (click *B.4.1.3*).

You will see the different fields of the form, each enclosed in a box with a thin gray border. Right-click on the field (anywhere within the enclosing box) that you want to modify and a pop-up menu will appear. Place the mouse over “**Change To**”, placed at the top of the pop-up menu, and a secondary pop-up menu will be displayed. The next screenshot shows the main and the secondary pop-up menus that you should see (note that the image has been cut and does not show all the options you will see in the main

pop-up menu):



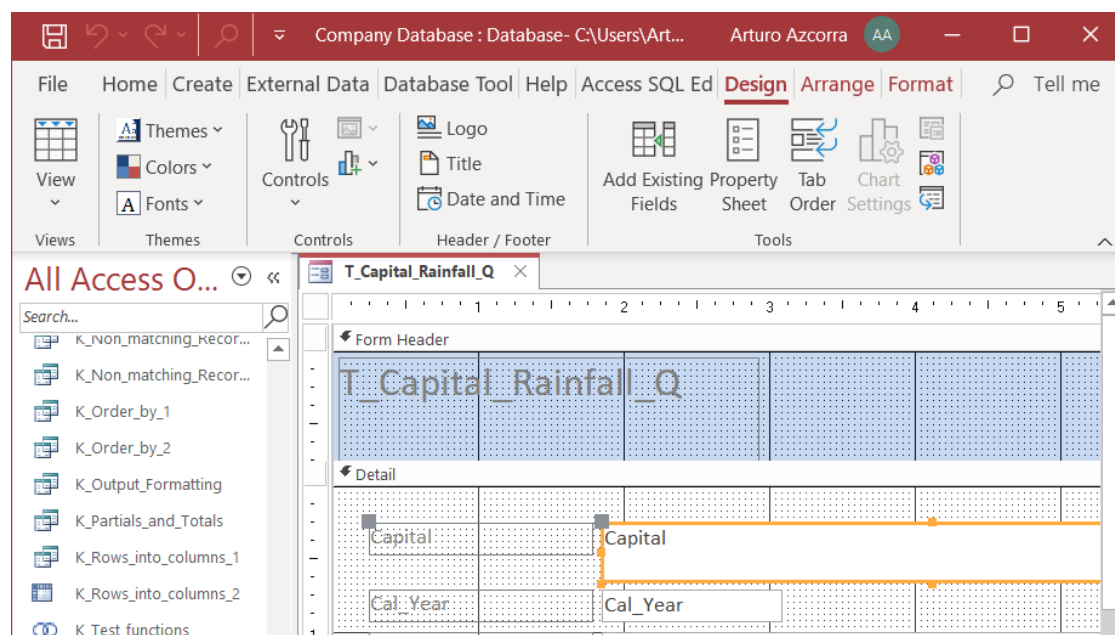
On the secondary pop-up menu, click on the “**Display Control**” that you want for this Form field. You can choose between “**List Box**” and “**Combo Box**”. My advice is you always select “**Combo Box**” because it is more flexible and has more configuration options than “**List Box**”.

Once you have selected the “**Display Control**” you want, it is required that you configure the “**Row Source Type**” and “**Row Source**” properties of this drop-down menu. You may also configure other additional properties. Click on *D.11.4.2* to find the different properties to be configured.

Setting a drop-down menu on Form fields does not affect stored data and has no side effects.

Notice that if instead of right-click you click on a field, the border of the box will change from thin gray to a thick yellow, and no pop-up menu will appear. This is not a problem, and you can right-click over the field that is highlighted with yellow border, and the same pop-up menu shown in the screenshot at the beginning of this subsection will appear. The following screenshot shows the field “**Num\_Byte**” highlighted with a

yellow border.



If you want to know how to configure different options in a **“Combo Box”**, you may click [“D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?”](#).

If you want to know some good practices in configuring your drop-down menu, you may click [“K.1.8 What are good practices in configuring my drop-down menus?”](#).

Once you are done with your Form configuration, **save** (click [B.4.1.6](#)) your Form. You may then **close** the Form (click [B.4.1.7](#)) or change it to **“Datasheet View”** (click [B.4.1.4](#)). If you get warning and/or **error** messages when saving your Form design, you may click [“L.2 How do I fix errors with my Table/Form design?”](#).

#### **D.11.4.2 How do I configure an existing drop-down menu in a Form field?**

If what you want is to configure a new drop-down menu in a Form field, you should read [“D.11.4.1 How do I configure a new drop-down menu directly in a Form field?”](#). Otherwise, you may continue reading here.

Regardless of the drop-down menu having been inherited from a record-source Table or having been set directly in the Form (as I explained in [D.11.4.1](#)), you can configure the drop-down menu by setting proper values for its properties.

If you want to change the type of **“Display Control”**, this is, you want to change between **“Text Box”**, **“List Box”** and **“Combo Box”**, you have to do it as if you were creating a new drop-down menu: this is explained in [“D.11.4.1 How do I configure a new drop-down menu directly in a Form field?”](#).

In case you decide to select **“List Box”**, you should be aware that in a **Table** it will **allow** you to input a value other than the ones in the list (e.g., typing it in, or editing the value selected from the list), but in a Form it **will not** allow you to input a value other than the ones in the list.

If you did not want to change the type of **“Display Control”** (or after having done it), you can configure the properties of the pop-up menu. You first unhide the **“Property Sheet”** in **“Design View”** (click [B.8.1](#)).

If you want to know all the available configuration properties and the effect of each of them you may click *“D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?”*.


Some of the properties you need to configure are placed under the **“Data”** tab, and others are placed under the **“Format”** tab, as follows:

Drop-down menu configuration properties under the **“Data”** tab:

- **“Row Source Type”** property (**mandatory**)
- **“Row Source”** property (**mandatory**)
- **“Bound Column”** property
- **“Limit to List”** property
- **“Allow Value List Edits”** property
- **“Allow Multiple Values”** property
- **“List Items Edit Form”** property
- **“Show Only Row Source Values”** property

Drop-down menu configuration properties under the **“Format”** tab:

- **“Column Count”** property
- **“Column Heads”** property
- **“Column Width”** property
- **“List Rows”** property
- **“List Width”** property

Remind that you can toggle between property default order and alphabetical order by clicking on the **A-Z**  icon placed at the top right corner of the **“Property Sheet”**. Remind also that you can see all the properties under the **“All”** tab.

Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form. You may then **close** the Form (click *B.4.1.7*) or change it to **“Datasheet View”** (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Form design, you may click *“L.2 How do I fix errors with my Table/Form design?”*.

Configuring a drop-down menu on Form fields does not affect stored data and has no side effects.


If you want to know some good practices in configuring your drop-down menu, you may click *“K.1.8 What are good practices in configuring my drop-down menus?”*.

#### **D.11.4.3 How do I configure a new date-picker directly in a Form field?**

You can **only** configure a date-picker in a field with **field type Date/Time**.

Unhide the Form **“Property Sheet”** in **“Design View”** (click *B.8.1*).

The **“Property Sheet”** shows at its top **an element box** with a drop-down menu to select the Form element for which you want to show its properties. **Select** the **name** of the field where you want to configure a date-picker.

To configure a date-picker, click on the **“Format”** tab (or the **“All”** tab) and find the **“Show Date Picker”** property. Remind that you can order properties alphabetically clicking on the **A-Z**  icon (top-right corner).

You click on the **rightmost** side of the cell placed **to the right** of the “**Show Date Picker**” property and click on “**For dates**” from the pop-up menu.

Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form. You may then **close** the Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Form design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

### **D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu?**

You may click:

- “*D.11.5.1 What are useful options in a “Combo Box” Table/Form drop-down menu?”*”
- “*D.11.5.2 What are the interactions between slave fields, “Limit to List” property and the field validation rule?”*”

#### **D.11.5.1 What are useful options in a “Combo Box” Table/Form drop-down menu?**

We have already seen the “**Combo Box**” properties “**Row Source Type**” and “**Row Source**” (click *D.11.1* or *D.11.4*). The **other** properties of “**Combo Box**” are used less frequently, yet some of them are quite useful, so I will now explain each of them:

- “**Bound Column**” property  
**Combo Box** column that contains value that control is set to. This is related to internal management of MS-Access. My advice is you always leave the default value of “**1**”.
- “**Column Count**” property  
The number of Table/Query columns shown in the drop-down menu. Showing a second column can be very useful if the values are not meaningful. For example, if you want a drop-down of product **codes**, it may be very useful to show a **second** column in which the **product name** is shown.
- “**Column Heads**” property  
Selecting “**Yes**” will show the Table/Query field name(s) as the heading of each column of the drop-down menu.
- “**Column Width**” property  
When you show more than one column from the drop-down menu, this allows to configure each column’s width. Column widths are indicated as a **list** of width values separated with **semicolon**. Each width value is a number followed by the length units. You may type-in one double quote character “”” to indicate that the units are inches, or you may type in “**cm**” to indicate that the units are centimeters. Regardless of what length units you type-in (“ or **cm**) MS-Access will rewrite them (doing the correct computation) to the units of the language version (inches for English version and centimeters for international versions).

If you put in the list less values than the number of columns, absent values will be interpreted as **zero**, and the corresponding columns **will not be shown**. An example of a column width expression for three columns is: “**0.8" ; 1.4" ; 1.1"**”.



- **“List Rows” property**

The number of value rows shown in the drop-down menu. If there are more values than shown rows, the drop-down menu will have a scrollbar on its right side.

- **“List Width” property**

Width of the complete drop-down menu indicated as one number followed by the length units. You may type-in one double quote character “” to indicate the units are inches, or you may type in “**cm**” to indicate the units are centimeters. Regardless of what length units you type-in (“ or **cm**) MS-Access will rewrite them (doing the correct computation) to the units of the language version (inches for English version and centimeters for international versions).

If the value of “**List Width**” is **larger** than the **sum** of the values of the “**Column Width**” property (see above), the width of the rightmost columns will be **truncated** to adjust. If the value of “**List Width**” is **smaller** than the **sum**, the width of the **rightmost** column will be made **wider** to adjust to the total “**List Width**”.

- **“Limit to List” property**

Setting it to “**Yes**” **will prevent** you from modifying the value you selected from the drop-down menu. Setting it to “**No**” will **allow you to edit** the value you selected from the drop-down menu and change it to a value not included in the menu. When the drop-down menu shows **all the possible values** for this field (e.g., list of calendar months), my advice is to configure this as “**Yes**”. However, if the drop-down menu only shows **some** of the possible values, you should configure this as “**No**”.

Notice that even if you set this property to “**Yes**”, you will be able to **paste** records with values other than the ones in the list! If you really want to limit the values that the field can contain, you will have to configure it as a **slave** field (click *D.9*) and/or write a suitable field validation rule (click *D.5.1.5*).

Recall that the properties listed in this section correspond to the “**Display Control**” value of “**Combo Box**”. In case you decided to select “**List Box**”, this property (and others) will not be available. Selecting “**List Box**” in a **Table** will **allow** you to input a value other than the ones in the list (e.g., typing it in, or editing the value selected from the list), but selecting it in a **Form** it **will not** allow you to input a value other than the ones in the list.

- **“Allow Multiple Values” property**

Setting it to “**Yes**” allows you to store a **list of values** (instead of the usual **single** value) in this field. Also, it will allow you to select **as many values as you want** from the drop-down menu. Each row from the drop-down menu will have a checkbox to its left. You can **tick** as many checkboxes as you want. When you are done, click on “**OK**” (at the bottom of the drop-down menu) and a list of **all** the selected values will be stored in the field. If you want to cancel entering this list of values, click on “**Cancel**” instead of on “**OK**”.

Be aware that the data type of every element of the list **must match** the field type configured for this field. Also be aware that in a value list you cannot have duplicate values.

If you configure “**Allow Multiple Values=Yes**” MS-Access **will not** allow you to **type-in** values in this field, **nor to edit** the list of values you selected from the drop-

down menu. This MS-Access behavior is equivalent to the one when you set “**Limit to List=Yes**”.

If you want to know more about storing a **lists of values** in a **field**, you may click “*K.2.8 How do I store a list of values, instead of a single value, in a Table field?*”.

- “**Allow Value List Edits**” property  
Setting it to “**Yes**” allows you to modify the values **of the drop-down menu itself** by using the Form indicated in the property “**List Items Edit Form**”.
- “**List Items Edit Form**” property  
This is the Form that you will use to edit the values in the drop-down menu itself, in case you configured “**Allow Value List Edits=Yes**”.
- “**Show Only Row Source Values**” property  
In case you have set “**Allow Multiple Values=Yes**”, setting this property to “**Yes**” will show only the values that match the current row source.

You should also be aware about some **interactions** between **slave** fields, the property “**Limit to List**” and the **field validation rule** (I present this in the next subsection).

Once you are done with all your Table/Form configuration, **save** (click *B.4.1.6*) your Table/Form configuration. You may then **close** the Table/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*). If you get warning and/or **error** messages when saving your Table/Form design, you may click “*L.2 How do I fix errors with my Table/Form design?*”.

#### **D.11.5.2 What are the interactions between slave fields, “**Limit to List**” property and the field validation rule?**



If you configure the drop-down menu of a **slave field** from the list of values of its **master field**, configuring it as “**Limit to List=Yes**” is somehow redundant, because MS-Access will anyway **not** allow you to enter a value that is not in the **master** field. However, I still think it is worth configuring “**Limit to List**” to “**Yes**” because the effort is negligible, and in this way, you will get the error warning right when you enter the value in this field, instead of when you have entered **all** the values in your new record.

If you configure “**Limit to List=Yes**”, you may think it is unnecessary to add a validation rule that checks that the field values are correct (e.g., that conform to the values in the list). However, you should know that when you **paste** over a rectangle of fields (or you paste as new records), MS-Access **will not check** if the pasted values exist in the drop-down menu. Therefore, you can **paste** values that **do not** exist in the drop-down menu. It is therefore convenient that you configure a field validation rule to check (when possible, which is not always the case) that the value exists in the drop-down menu. If you want to know more about field validation rules, you may click “*D.5.1.5 What is the field “Validation Rule” Table field property?*”.

## **D.12 How do I use MS-Access Reports?**

MS-Access **Reports** are objects that allow **presenting** data in very flexible way from **one or more** Tables or Queries.

To create a Report over several Tables/Queries, click on the “**Create**” Ribbon, and then

on the **Report Wizard**  icon, placed in the top right corner of the “**Reports**” Ribbon group. In the “**Report Wizard**” dialog box click on the **drop-down menu**  icon and click on the first Table/Query you want the Report to be based on. Then double-click on each field from the Table/Query that you want to incorporate to the Report. When you are done with the fields of this Table/Query, you select another Table/Query using the drop-down menu at the top of the Wizard. You do this for as many Table/Queries you want. When you are done incorporating fields from your Table/Queries, click on “**Next**”. You then click on “**Datasheet**”. You can now type-in the name you want for this Report in the text box at the top of the Wizard window. Then click on “**Finish**” and the Report will be created.

### **D.13 How do I share a database, having multiple concurrent users?**

You **share** a database setting the option “**Default open mode**” to “**Shared**” and placing the database **file** in a **network drive** that all the users can access.

To configure the sharing option, click on the sequence “**File**→**Options**→**Client Settings**→**Advanced**→**Default open mode**” and set the option to “**Shared**”. If you rather set it to “**Exclusive**”, then the MS-Access file is configured to be opened **by only one** user and cannot be shared. The default setting is “**Shared**”, but you can check its value if you want.

If an MS-Access database **file** is configured as “**Shared**” and placed in a network drive, **several** users can **simultaneously** open the file and work over the database in parallel. Concurrent users can invoke Queries, modify records, and do any operation they may need over the database. In case the network connection from one user to the network drive is poor, then **this** user will experience poor performance in the Queries and other operations he/she performs over the database, but the other ones will have good performance.

What I have described here is **only** the **basics** of sharing an MS-Access database. If you want more **advanced design** advice, you may click:

- “*K.3 How do I structure and optimize a distributed database?*”

## PART E. ENTERING, MODIFYING AND DELETING MY DATABASE DATA

You can **change** Table records by **entering**, **modifying** or **deleting** them.

You can **enter**, **modify** and **delete** Table records from the **Table** in “**Datasheet View**” (click *B.5*) or from a **Form** (in “**Datasheet View**”, click *B.5*) derived from a Table. The way to **change** Table records from a Table or Form is **almost** the same: you may see the few differences clicking *E.4*.

You can **also enter**, **modify** and **delete** Table records from a **Query result** in “**Datasheet View**” (click *B.5*), but this is considered a **bad practice** and should be **prevented**. If you want to know how to **prevent** this, you may click “*K.4.4 Why should I disable changing Table data from Query results?*”. Since changing records from a Query result in “**Datasheet View**” is a bad practice, I am not mentioning this feature along this **Lightning Guide**, and I have only addressed it in the said section *K.4.4*.

You can **enter new** Table **records** in the following ways:

- **Editing one new** record:  
Click “*E.1 How do I edit one new or existing record?*”.
- **Pasting as new** records:  
Click “*E.5.2.3 How do I paste as new records?*”.  
Click “*E.7.2 How do I bulk-change my data with an external application?*”.
- **Inserting** records with an SQL Query:  
Click “*F.13.2 What is an Insert operation and how do I write it?*”.

You can **modify existing** Table **records** in the following ways:

- **Editing one existing** record:  
Click “*E.1 How do I edit one new or existing record?*”.
- **Pasting** values over fields of **existing** records:  
Click “*E.5.2.2 How do I paste over a rectangle of fields?*”.  
Click “*E.7.2 How do I bulk-change my data with an external application?*”.
- **Updating** existing records with an SQL Query:  
Click “*F.13.3 What is an Update operation and how do I write it?*”.
- **Searching and replacing** existing records:  
Click “*E.7.1 How do I bulk-modify my data using the “Find/Replace” tool?*”.
- **Cascade updating** slave fields in slave records in Relationships with referential integrity:  
Click “*D.9.4 What is the effect of “Cascade Update Related Fields”?*”.

You can **delete existing** Table **records** in the following ways:

- Interactively **deleting** existing records:  
Click “*E.3 How do I interactively delete existing records?*”.
- **Deleting** existing records with an SQL Query:  
Click “*F.13.1 What is a Delete operation and how do I write it?*”.
- **Cascade deleting** slave records in Relationships with referential integrity:


Click “D.9.5 What is the effect of “Cascade Delete Related Records”?””.


You may also click directly on:

- “E.1 How do I edit one new or existing record?”
- “E.2 How do I edit the field’s values of the record under edition?”
- “E.3 How do I interactively delete existing records?”
- “E.4 What is different about entering, modifying or deleting records from a Table or a Form?”
- “E.5 How do I copy/cut and paste data between MS-Access and other applications?”
- “E.6 What checks are done when saving a field value, or entering or modifying a record?”
- “E.7 How do I bulk-change my Table/Form’s data?”
- “E.9 Can I get inconsistent results out of my initial data in my database?”
- “E.10 Why should I use “Compact and Repair Database”?”

## **E.1 How do I edit one new or existing record?**


Open the Table/Form in “**Datasheet View**” (click B.4.1.3).

If you want to **edit** one **new record**, go to the **new-record row** (click E.1.1) and **start to edit** (see below) one (or more) of the field values of the **new-record row**. Right then, the **new-record row becomes a new record under edition** (marked with the **editing “”** icon on its left) and the **new-record row appears right below the new record under edition**.

If you want to **edit** one **existing record**, find the **record** that you want to **edit** (click B.5.5) and **start to edit** (see below) one (or more) of its field values. Right then, the record **becomes an existing record under edition**, marked with the **editing “”** icon on its left.

You **start to edit** (as required above) one (or more) of the field values in either of the following ways:

- **Select** one **field’s value** (click B.5.3), within the **new-record row** or within an **existing record**, and **type-in a first character** (or press either the “Delete” or “Supr” keys). Notice that when you **select a field’s value** the value **shown may change** (click E.2.3).
- **Choose** a field value with its drop-down menu, date-picker, or checkbox, within the **new-record row** or within an **existing record**. This will also **select the field’s value**.
- **Successfully paste** (click E.5.2.2) **value(s)** over one (or more) **field(s)**, within the **new-record row** or within an **existing record**. This will also **select the field’s value** of the leftmost field over which you pasted.

Regardless of the way you start to edit, the **new-record row** or the **existing record** becomes at that moment the **record under edition**, marked with the **editing “”** icon on its left. You may now **edit** (click E.2) the field values of the **new** or **existing record under edition**.

Once you are done editing its field values, you **enter** (click *E.1.2*) the **record under edition**.

For the case of editing **one new** record, if one (or more) **field(s)** have a “**Default Value**” in the Table design (click *D.5.1.4*), you will see the corresponding **default value** shown for each such **field** in the **new-record row**. Also, if one (or more) **field(s)** have an **event value** in the Form design (click *D.10.4*), you will see the corresponding **event value** shown for each such **field** in the **new-record row**.

Notice MS-Access **locks** the **record under edition**, to prevent that more than one concurrent user can edit it at the same time (click *K.3.14*).

Once you have finished **editing new** or **existing** records, you may **close** the Table/Form (click *B.4.1.7*).

You may also click:

- “*E.1.1 How do I go to the new-record row?*”
- “*E.1.2 How do I enter the record under edition?*”

### **E.1.1 How do I go to the new-record row?**

Open the Table/Form in “**Datasheet View**” (click *B.4.1.3*).

The **new-record row** is the **downmost**<sup>22</sup> row of the Table/Form in “**Datasheet View**”, and it has a small cell with an **asterisk** “\*” icon on its leftmost side (see slightly below how to go to it). You **go to** the **new-record row** in either of the following ways:

- Click on the **new-record** “\*” icon from the navigation-bar (at the very bottom of the “Table/Form pane”). This will take you to the **new-record row** and will **also** select the **field’s value** of its **leftmost** field.
- Right-click on any of the small gray cells on the left side of the “Table/Form pane” and click on “**New Record**” from the pop-up menu. This will take you to the **new-record row** and will **also** select the **field’s value** of its **leftmost** field.
- Click on “**Home**” from the Ribbon-bar. Click on the **Go to** “→ v” icon from the Ribbon and then click on “**New**” from the pop-up menu. This will take you to the **new-record row** and will **also** select the **field’s value** of its **leftmost** field.
- Click on “**Home**” from the Ribbon-bar. Click on the **Go to** “→ v” icon from the Ribbon and then click on “**Last**” from the pop-up menu. This will take you to the end of the Table/Form where you will **see** the **new-record row**.
- Click on the last-record “▶|” icon from the navigation-bar (at the very bottom of the “Table/Form pane”). This will take you to the end of the Table/Form where you will **see** the **new-record row**.
- Scroll down in the Table/Form until you **see** the **new-record row** (if the Table/Form is large this may be tedious).

For the case of the **first** three bullets right above, you will be in the **new-record row** and you will **also** have **selected** the **field’s value** of its **leftmost** field. For the case of

---

<sup>22</sup> Except if you are using an aggregation row in the Table/Form, in which case the “new-record row” is right above the aggregation row (which is the very last row).

the **last** three bullets right above, you will **not** be in the **new-record row**: you will just be **seeing** it. In either case you will most likely want to **edit** a **new record**: click “*E.1 How do I edit one new or existing record?*”.

Notice that if one (or more) **field(s)** have a “**Default Value**” in the Table definition (click *D.5.1.4*), you will see the corresponding **default value** shown for each such field in the **new-record row**. Notice that if one (or more) **field(s)** have an event value in the Form definition (click *D.10.4*), you will see the corresponding **event value** shown for each such field in the **new-record row**.

### **E.1.2 How do I enter the record under edition?**

You indicate that the **record under edition** should be **entered** into the Table in either of the following ways:

- **Selecting a field’s value** (click *B.5.3*) in **another** record.
- **Selecting** one (or more) **fields** (click *B.5.2*).

If you were **editing** a field value, when you do either of both actions, you will **also** indicate MS-Access that the field value under edition should be **saved**. In this case, MS-Access will first attempt to **save** the field value (click *E.2*) **before** attempting to **enter** the record.

Else, (if you were **not** editing a field value or you were, but all its field value error checks have been correct), MS-Access will perform several **record error checks** over the **record** (click *E.6.2*), with the following results:


- **If one (or more) of these error checks is wrong**  
MS-Access will show an **error box** indicating the cause of the error, and the **record under edition** will remain **selected** (i.e., the record or fields that you had clicked on will **not** be selected).

Click on “**OK**” to remove the error message. Then, you **fix** the error depending on which error message you got (click *L.4.3*). You may rather **cancel** the **editing** of the record by pressing the “**Esc**” key. If you cancel, you have two cases:

- For a **new record under edition**: the record itself is **lost**, and a field in the **new-record row** remains selected.
- For an **existing record under edition**: all the edited field values are lost, the **existing** record remains **unmodified** in the Table, and a field of the record under edition remains selected.

Notice that MS-Access will **not allow you do anything else in this Table** until you either **correct** the error(s) or **cancel** the editing of the record.

- **If all the record error checks are correct (or you already corrected all reported errors)**  
The record is **entered**, and the other record (or fields) that you had selected to indicate that the **record under edition** should be **entered** will now be selected.

Either if you **cancelled** the **editing**, or the record was successfully **entered**, the record is now **not under edition**. You can see this because the **editing** “” icon that was shown in the narrow column to the left of the **record under edition** is not there anymore.

If this is a **master** Table in one (or more) Relationship(s) (click *C.11*) with referential integrity (and “**Cascade Update Related Fields**” set, click *D.9.4*), the new values in **this record’s master** fields will be automatically updated in the corresponding **slave** fields of **all** its **slave** records.

## **E.2 How do I edit the field’s values of the record under edition?**

You **edit** the field values of the **record under edition** in either of the following ways.

- **Typing-in** the field value  
Select one **field’s value** (click *B.5.3*) and **edit** the value typing-in with the keyboard (click “*E.2.2 How do I type-in a value in a field?*”). Notice that when you **select a field’s value** the value **shown may change** (click *E.2.3*).
- **Choosing** the field value  
**Choose** the field value with its **drop-down menu**, **date-picker**, or **checkbox** : click “*E.2.1 How do I choose a field value?*”. This only applies if the field has been one configured **one** of them. Notice that you can first **choose** a value and then **edit** it by typing-in (if the field configuration allows for this, click *K.1.8.3*).

If the field has a drop-down menu, it is usually much better to **choose** the field value using the menu instead of **typing-in** a value: a drop-down menu is faster, and you avoid typing errors.

Once you are done editing the field value, you indicate that it has to be **saved** in either of the following ways:

- **Selecting another field’s value** of the **record under edition** (click *B.5.3*)  
This simultaneously indicates MS-Access to **save** the currently edited field value, and that you **continue** editing the **record under edition**.
- Attempting to **enter** the **record under edition** (click *E.1.2*).  
This simultaneously indicates MS-Access to **save** the currently edited field value, and that you want to **enter** the **record under edition** into the Table.

Regardless of the way you do it, MS-Access will perform several **field value error checks** (click *E.6.1*), with the following results:

- **If one (or more) of these error checks is wrong**  
MS-Access will show an **error box** indicating the cause of the error, and the **field’s value** of the field under edition will remain **selected** (i.e., the other field where you had clicked will **not** be selected).

Click on “**OK**” to remove the error message. Then, you **fix** the error depending on which error message you got (click *L.4.2*). You may rather **cancel** the **editing** of the field value by pressing the “**Esc**” key. If you cancel, the field value will remain as before you started to edit it, and you will remain in the currently edited field (i.e., you are **not** moved to another field).

Notice that MS-Access will **not allow you do anything else in this Table** until you either **correct** the error(s) or **cancel** the editing of the field value.

- **If all the field error checks are correct (or you already corrected all reported**



errors)

The field value is **saved**, and the other field that you had selected to indicate that the currently edited field value should be **saved** will now be selected.

When editing the **field's values** of a **new record under edition** it is quite convenient to start editing the **field's value** of its **leftmost field**, and then continue editing the other **field's values one by one left to right**. A practical way to do this (as indicated above) is by pressing the “**Enter**” or the “**Tab**” key. Pressing either key will simultaneously attempt to **save** the current **field's value** (in case you were editing it) and **select** the next **field's value** on the right.

For the case of **Calculated** fields, be aware that their values **cannot** be **edited** (**and it is not actually stored**): its value is **calculated** from the other field values of each record, each time that the **Calculated** field is used.

Finally, be aware that MS-Access **may change the value you edited** in the field and **store a different** one. I am not referring to changing the way the value is **shown**, but to actually **changing the stored value**. For example, for **Short Text** fields MS-Access will **remove any trailing space characters** that you had edited. Another example, for **Integer** and **Long Integer** fields MS-Access will do round-half to even (click *J.11.20*) on any **fractional** number that you edit into the field. If you want to know more about why and how MS-Access may **change the values** you edit into a field, you may click “*L.4.2 How do I fix error messages when saving a field value?*”. If you want to know more about rounding problems, you may click “*J.11.20 How do I fix a Query making rounding errors?*”.

Notice further that even if the stored value is the one you edited into the field, MS-Access may **show a different** one. The reason is that the value will be **shown** according to the “**Format**” property of this field (click *H.6*).



You may click:

- “*E.2.1 How do I choose a field value?*”
- “*E.2.2 How do I type-in a value in a field?*”
- “*E.2.3 Why is a value shown in a different way when the Table/Query/Form field's value has been selected?*”






### **E.2.1 How do I choose a field value?**

Open the Table/Form in “**Datasheet View**” (click *B.4.1.3*).

You can **choose** a field value within **one** record in either of the following ways:

- **Using its drop-down menu** (in case the field has it)  
Click on the rightmost side of the field (cell) and click on the value you want among the options shown in the drop-down menu. If the drop-down menu has more options than can be shown, you can scroll the values with the menu's scrollbar until you find the one you want, and then click on it. Notice that the **drop-down menu** “” icon is **only** shown if the **field's value** has been selected. However, even if the “” icon is **not shown**, if you click on the rightmost side of the field/cell, **the drop-down**

menu will appear.

- **Using its date-picker** (in case the field has it)  
Select the **field's value** (click *B.5.3*) and the date-picker “” icon will be shown to the right of the field. If you click on the icon, a pop-up date-picker (a calendar) will appear, where you can **choose** a date by clicking on the day you want. You can change the month that is shown by clicking on the **previous** “” icon or **next** “” icon at the top of the date-picker. A date-picker can only be configured in **Date/Time** fields.
- **Using its checkbox** (in case the field has it)  
You can **change** the field's value by clicking on the checkbox. If the field's value is **True/Yes/On** or **ticked** a **ticked checkbox** “” is shown. Otherwise, an **unticked checkbox** “” is shown. Notice that if the field has a checkbox you **cannot type-in** the value, and you can only **choose** it using the checkbox. A checkbox can only be configured in **Yes/No** fields.

### E.2.2 How do I type-in a value in a field?

Select the **field's value** (click *B.5.3*). When you select the **field's value**, the value **shown** may change (click *E.2.3*).

In case the field is **Null** or contains either the **zero-length** string or an **invisible** string (click *L.7.8*) the field will be blank.

Otherwise, you will see the **string** representing the field value. If the string is larger than what fits in the field size, you will only see part of the string. You may see part (or all) of the string in **white** font over **black** background: this indicates that this part of the string is selected. You may also see the type-in cursor, shown as a vertical bar “|” (blinking for a while) inside the field.

You may now edit the **string** representing the field value using the keyboard and the mouse, as you would do with a conventional text editor:

- **Select** part of the **string** doing double click or click-and-drag (the selected part will be in **white** font over **black** background).
- **Delete string** characters where the type-in cursor “|” is placed pressing the “**Del**” or “**Supr**” keys.
- Add characters to the **string** where the type-in cursor “|” is placed by typing them in.
- **Overwrite** the **selected** part of the **string** that is **selected** by pressing any character key.
- **Delete** the **selected** part of the **string** by pressing any character key.
- **Move** the type-in cursor “|” along the **string** by pressing the left-arrow “**←**” key or the right-arrow “**→**” key, or clicking with the mouse in the corresponding place of the **string**.
- **Cut** or **copy** the **selected** part of the **string**.
- **Paste** as **text** a string previously copied (or cut) elsewhere (click *E.5.2.1*).

The format of values that you type-in a field are **almost** the same as the **constants** you

**write** for expressions (click G.4). The **main** difference is that when **writing constants** the *Strings* must be enclosed in **quotes** and **Date/Time** constants must be enclosed between “#” characters. However, when **typing-in** a value, strings are typed-in **without** quotes and **Date/Time** values are typed-in without “#” characters.

Typing-in a value has some peculiarities depending on the value’s data/field type. You may click:

- “E.2.2.1 How do I type-in a value in a Yes/No field?”
- “E.2.2.2 How do I type-in a value in a Short Text field?”
- “E.2.2.3 How do I type-in a value in a Number, Currency or Large Number field?”
- “E.2.2.4 How do I type-in a value in a Date/Time field?”
- “E.2.2.5 How do I type-in a zero-time value in a Date/Time field?”
- “E.2.2.6 How do I type-in a zero-date value in a Date/Time field?”
- “E.2.2.7 What happens if the field “Format” property does not match the field type?”

#### **E.2.2.1 How do I type-in a value in a Yes/No field?**

To store “-1” (which represents **True/Yes/On**), you **type-in** either “**Yes**”, “**True**”, “**On**” (case insensitive) or **any number** other than “**0**”. To store “**0**” (which represents **False/No/Off**), you **type-in** either “**No**”, “**False**”, “**Off**” (case insensitive) or the number “**0**”.

If you are using a foreign-language version of MS-Access, you may click “L.8.12 How do I fix foreign-language issues of MS-Access?”.

#### **E.2.2.2 How do I type-in a value in a Short Text field?**

You just type-in the text string you want to store. Remind the following relevant questions:

- MS-Access will remove all trailing invisible characters (e.g., blanks) that you type-in
- If you want to enter a **line-feed** (which I strongly advice **not to do**) into the string, you can do it by pressing “**Ctrl-Enter**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**Enter**” key).

#### **E.2.2.3 How do I type-in a value in a Number, Currency or Large Number field?**

Type-in a positive or negative number in decimal notation or in scientific notation. You **cannot** type-in a character for separation of thousands.

Some examples of number values you can **type-in** are: **230**, **4.0346**, **730**, **-325.234**, **235.67E26**, **.258E-15** and **-12.87E+25**.

#### **E.2.2.4 How do I type-in a value in a Date/Time field?**

Typing-in a **Date/Time** value is much trickier than any of the other values. To start with,

it is very important to point out that a **Date/Time** field **always** stores **both** a **date** (the **date-part**) **and** a **time** (the **time-part**). Even if you type-in a zero-time value (i.e., typing-in only a date) or a zero-date value (i.e., typing-in only a time), the stored value **always contains both date and time**. If you **type-in** a **zero-time** value, it will be **stored** with the time-part **0:00:00**. If you **type-in** a **zero-date** value, it will be **stored** with the **default** date-part **30-december-1899**. Even though the value is **always** stored with **both** a date-part **and** a time-part, you can configure the “**Format**” property to **show** both date and time, to **show** only the date-part or to **show** only the time-part (click *H.6*).

You type-in a **Date/Time** value by typing-in either:

- A zero-time value (click *E.2.2.5*)  
For example: typing-in “**3-jan-2014**”, “**3-1-2015**” or “**3-jan**”.
- A zero-date value (click *E.2.2.6*)  
For example: typing-in “**13:24:35**” or “**13:36**”.
- **Both** a zero-time value **and** a zero-date value, in **either order**, and separated by one (or more) blank characters.  
For example: typing-in “**3-jan-2014 13:24:35**” or “**13:24:35 3-jan-2014**”.

MS-Access allows you to type-in a **zero-time** value and a **zero-date** value in **many ways**. If you want to know how, you may click:

- “*E.2.2.5 How do I type-in a zero-time value in a Date/Time field?*”
- “*E.2.2.6 How do I type-in a zero-date value in a Date/Time field?*”

If you are using a foreign-language version of MS-Access, you may click “*L.8.12 How do I fix foreign-language issues of MS-Access?*”.

### **E.2.2.5 How do I type-in a zero-time value in a Date/Time field?**

You can type-in a zero-time value as **day, month and year** separated by either “/”, “-” or “,”. My advice is you always type-in zero-time values as day, month name prefix and four-digit year, to avoid ambiguity and confusion. However, for completeness’ sake, I will now detail all the different ways in which you can write a zero-time value.

You can also type-in a zero-time value as **day and month**, in which case the year will be the **current year**.

You can also type-in a zero-time value as **month and year**, in which case the day will be **1**.

You **cannot** type-in a zero-time value as **day and year**.

You can type-in **the month** as a **month number** or as any **prefix** of the **month name** down to a minimum of three characters. You can type the month name prefix **in any case** (it is case insensitive).

You can type-in **the year** as four digits, three digits or two digits. If you type-in the year as **three or four digits**, then that is the year. If you type-in the year as **two digits**, they are interpreted as the **last two** digits of the year. If the two-digit year is less than or equal to 49, the first two digits of the year are “20”. If the two-digit year is greater than or equal to 50, the first two digits of the year will be “19”.

You can type-in the day, month and year **almost in any order**, and MS-Access will

correctly interpret each of them, as long as there is no ambiguity. For example, if you type a four-digit year, a day number greater than 12 and a month number, regardless of the order, it is clear which is which. Also, if you type-in a four-digit year, a month name and a day, there is no ambiguity. When there is ambiguity, MS-Access will apply the following rules:

- **Both numbers less than or equal to 12**  
The first number is interpreted as month number and the second as day number. The year will be the current year.  
Some examples are “**2-3**” (interpreted as Feb-3-current\_year) and “**3-2**” (interpreted as Mar-2-current\_year).
- **One number less than or equal to xx, the other number less than or equal to 12**  
where xx is the number of days of the month corresponding to the other number. The number less than or equal to xx is interpreted as day number, the other number as month number. The year will be the current year.  
Some examples are “**2-25**” or “**25-2**”, both interpreted as Feb-25-current\_year.
- **One number greater than xx, the other number less than or equal to 12**  
where xx is the number of days of the month corresponding to the other number. The number greater than xx is interpreted as the last two digits of the year, and the other number as the month number. The day is “1”.  
Some examples are “**4-34**” and “**34-4**”, both interpreted as Apr-1-2034). Other examples are “**4-50**” and “**50-4**”, both interpreted as Apr-1-1950.

My advice is **avoid entering ambiguous date values** to prevent unintended errors.

Because MS-Access accepts the “,” as separator of date elements, notice there is a slight risk of mistakenly entering a number in a **Date/Time** field in countries where “,” is also used for decimal separation.

Notice that in non-English versions of MS-Access the translated complete month names will also be accepted, in addition to the English month name prefix (down to the first three letters). If you are using a foreign-language version of MS-Access, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

To clarify all the rules above, I am including below some additional examples.

- Month (number or name prefix), day number and year number (with two or four digits), separated by either “/”, “-” or “,”. If you use only two digits for the year, the two first year digits are interpreted as “20”.  
Some examples are: **3/30/2017**, **12/31/87**, **3-30-2017**, **12-31-87**, **march/30/2017**, **December/31/87**, **mar-30-2017**, **dec-31-87**, **MARCH/30/2017** and **Decem,31,87**
- Year number with **four digits**, month (number or name prefix) and day number, separated by “/” or by “-” or by “,”.  
Some examples are: **2017-12-30**, **2017/12/30**, **2017-dec-3**, **2017,MARCH,3** and **2017/April/3**
- Month (number or name prefix) and day number separated by either “/”, “-” or “,”. The year stored will be the current year.  
Some examples are: **3/30**, **12/3**, **3-30**, **12-31**, **march/30**, **December/31**, **mar-**

**30, dec-31, MARCH/3 and Decem,31**

- Day number and month name prefix separated by either “/”, “-” or “,”. The year stored will be the current year.  
Some examples are: **30/march, 31/DECEM, 3-30-april** and **31,February.**
- Day number higher than 12 and month number separated by either “/”, “-” or “,”. The year stored will be the current year.  
Some examples are: **30/3, 31/12, 13-10** and **31,6.**
- Year number with **four digits** and month (number or name prefix), separated by “/” or by “-” or by “,”. The day stored will be 1. Some examples are: **2017-12, 2017/12, 2017-dec, 2017,MARCH** and **2017/April**
- Month (number or name prefix) and year number with **four digits**), separated by “/” or by “-” or by “,”. The day stored will be 1.  
Some examples are: **12-2017, 12/2017, dec-2017, MARCH/2017** and **April/2017**

Remind once again that you may type-in a zero-time value, but the **Date/Time** field **always** stores **date and time**. If you enter a zero-time value, the time that it will be stored is 0:00:00, which corresponds to internal fractional value zero.

If you are using a foreign-language version of MS-Access, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

**E.2.2.6 How do I type-in a zero-date value in a Date/Time field?**

You can type-in a zero-date value as **hour, minutes and seconds** separated by either “:” or “.”. You can also type-in a zero-date value as **hour and minutes** (stored seconds will be zero). The order **must be** hour first followed by minutes, and optionally followed by seconds. Hour must be between 0 and 23 (both included). Minutes and seconds must be between 0 and 59 (both included). Notice the time 24:00:00 **does not exist**.

You can optionally append the string “**am**” or “**pm**” (case insensitive), separated by zero or more blanks. Adding the string “**PM**” to a zero-date value with an hour between 0 to 11 (both included) is equivalent to adding 12 to the hour value. Adding the string “**PM**” to a zero-date value with an hour greater than or equal to 12 has no effect.

You can also type-in a zero-date value as only hour (e.g., a value between 0 and 23) as long as you append the string “**am**” or “**pm**” (case insensitive), separated by zero or more blanks.

Examples of zero-date values are: **11:59, 23am, 23.59.46, 23.59.46 AM, 3:59pm** and **3:59:46 PM**

If you are using a foreign-language version of MS-Access, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

**E.2.2.7 What happens if the field “Format” property does not match the field type?**

The **values** that can be **correctly entered** and **stored** in a field **depend** on the **field type**, and **not** on the field **formatting**. If the “**Format**” property does not match the field type, you will **enter** and **store** the values according to the **field type**, but the values will be

**shown** (even in the editing formatting) according to the “**Format**” property.

I illustrate this with some examples:

- If you **format** a **Number** field as a date, as a time or as a date and a time, this will change the way the value is **shown**, but it will **not** change the fact that you can only introduce number values in this field.
- If you **format** a **Date/Time** field as zero-time, this will change the way the value is **shown**, but it will **not** change the fact that the field **always** stores a **date-part** **and** a **time-part**, and you can therefore introduce **date and time** values in this field.
- If you **format** a **Number** field as **On/Off**, this will change the way the value is **shown**, but you will still be able to **store** any number other than “0” and “-1”, which you **cannot** do in a **Yes/No** field.

### **E.2.3 Why is a value shown in a different way when the Table/Query/Form field’s value has been selected?**

First of all, it is important to point out that the **actual stored value** does not change at all, it is just being **shown** in a different way.

When you see a field value in a Table/Query/Form in “**Datasheet View**” the value is shown with a **formatting** according to the “**Format**” property. However, if you **select the field’s value** (e.g., by clicking within the field, click *B.5.3*), MS-Access will show you the value **with the editing formatting**, according to the internal rules of MS-Access. The **formatting** of the “**Format**” property and the **editing formatting** may be different, and therefore, you notice that when you select a field value, **the value that is shown may change**. Even though Query output fields resulting from an expression **cannot** be edited, when you select a Query’s field value (e.g., you click on one field), the value will be shown **with editing formatting** and **not** with the **formatting** of the “**Format**” property.

The possible differences between **formatting** of the “**Format**” property and the **editing formatting** depend on the field-type, the field value and on the value of the “**Format**” property. You may click on the following subsections to check a number of relevant cases:

- “*E.2.3.1 How is the formatting of a Number or Currency field changed when I select its field’s value?”*”
- “*E.2.3.2 How is the formatting of a Yes/No field changed when I select its field’s value?”*”
- “*E.2.3.3 How is the formatting of a Date/Time field changed when I select its field’s value?”*”

#### **E.2.3.1 How is the formatting of a Number or Currency field changed when I select its field’s value?**

When you **select** a **field’s value** (e.g., by clicking **within** the field, click *B.5.3*), the **actual stored value** does **not** change at all, it is just being **shown** in a different way.

Some cases of differences between the “**Format**” property **formatting** and the **editing**

formatting in **Number** or **Currency** fields are the following:

- A number field will be shown with the number of decimal digits indicated in its “**Decimal Places**” property. When you select the **field’s value** (e.g., by clicking **within** the field, click *B.5.3*) the value shown will display all the decimal digits.
- A number field may be shown as “**Yes**”, “**True**” or “**On**” because its “**Format**” property has been configured as **Yes/No formatting**. When you select the **field’s value** (e.g., by clicking **within** the field, click *B.5.3*) the value shown changes **to the number actually stored in the field**.
- A number field may be shown as “25/03/1900 6:00:00” because its “**Format**” property has been configured as “**General Date**”. When you select the **field’s value** (e.g., by clicking **within** the field, click *B.5.3*) the value shown changes **to the number actually stored in the field**.

### **E.2.3.2 How is the formatting of a Yes/No field changed when I select its field’s value?**

The **actual stored value** does not change at all, it is just being **shown** in a different way.

- If the value shown is **True/Yes/On**, when you select the **field’s value** “-1” will be shown.
- If the value shown is **False/No/Off**, when you select the **field’s value** “0” will be shown.

### **E.2.3.3 How is the formatting of a Date/Time field changed when I select its field’s value?**

The **actual stored value** does **not** change at all, it is just being **shown** in a different way.

Remind that a **Date/Time** format is stored internally same as **Double** floating-point number, where the integer part stores the date and the fractional part stores the time. **Integer zero** represents the date **30-dic-1899**. **Fractional zero** represents the time **00:00:00**.

Some cases of differences between the “**Format**” property **formatting** and the **editing formatting** in a **Date/Time** field are the following:

- If the stored date-part is **not 30-dic-1899** and the stored time-part is **not 00:00:00**, the **editing formatting** will be **date and time**, regardless of the value of the “**Format**” property. If the “**Format**” property is some form of date and time, the editing formatting will be the same. If the “**Format**” property is zero-time or zero-date, the editing formatting be: **mm/dd/yyyy hh:nn:ss**, all in numbers with 24 hour format.
- If the stored date-part is **30-dic-1899** and the stored time-part is **not 00:00:00**, the editing formatting will be **zero-date**, regardless of the value of the “**Format**” property.
- If the stored date-part is **not 30-dic-1899** and the stored time-part is **00:00:00**, the



**editing formatting** depends on the “**Format**” property as follows:

- If the “**Format**” property is some form of **date and time**, the **editing formatting** will also be **date and time**.
- If the “**Format**” property is some form of **zero-time** or of **zero-date**, the **editing formatting** will be: **mm/dd/yyyy** all in numbers.
- In the particular case where the stored date-part is **30-dic-1899** and the stored time-part is **00:00:00**, the **editing formatting** depends on the “**Format**” property as follows:
  - If the “**Format**” property is some form of **date and time** or of **zero-date** the **editing formatting** will **zero-date**.
  - If the “**Format**” property is some form of **zero-time** the **editing formatting** will also be **zero-time**.

### **E.3 How do I interactively delete existing records?**

Open the Table/Form in “**Datasheet View**” (click *B.4.1.3*).

Find the **record** that you want to **delete** (click *B.5.5*) and **delete** it in either of the following ways:

- Right-click on the small gray cell on the left side of the record you want to delete and click on “**Delete Record**” from the pop-up menu.
- **Select** the **range** of records you want to delete by doing either **click-and-drag** or **Shift+select** (click *B.5.2*). You then either:
  - Press the “**Supr**” key.
  - Right-click anywhere in the range of selected records and click on “**Delete Record**” from the pop-up menu.

Regardless of the way you deleted the record(s), MS-Access will show a warning confirmation asking you to confirm (**Yes** or **No**) the **delete** operation.

If MS-Access does not allow you to delete the record, you may click *L.4.5*. It may also be possible that the **source** file (where this Table is located) is **read-only**.

Once you have finished deleting records, you may close the Table/Form (click *B.4.1.7*).

### **E.4 What is different about entering, modifying or deleting records from a Table or a Form?**

Entering, modifying or deleting Table records from a Form in “**Datasheet View**” (click *B.5*) is **almost** the same as doing it from the Table itself in “**Datasheet View**” (click *B.5*). The only differences are the following:

- When you are **typing-in** a string in a **Short Text** field, and you press the “**Enter**” key, MS-Access allows you to enter multi-line text!!!! (in a Table, pressing the “**Enter**” key jumps to the next field). Tab does move you to the next field. Notice that there is an option/property to change this behavior to the same behavior as in Tables.

- Forms usually include **subroutines** (bound to **Events**, click *D.10.4*) that automatically **edit** a value into one (or more) fields. The value(s) is(are) calculated by the subroutines taking as input the value(s) of other field(s). Therefore, when you **save** the value of one field, its **Event** subroutine may cause that the values of one (or more) **other** fields **automatically appear** or **change**.

## **E.5 How do I copy/cut and paste data between MS-Access and other applications?**

You can **copy/cut** data from many other programs and **paste** it into an MS-Access **Table/Form**. You can also **copy/cut** data from a MS-Access **Table** or **Query** result and **paste** it into many other programs, an also in MS-Access.

You will be doing a lot of copy/cut and paste from other programs when you create a database, because you will want to **populate** the database Tables with **initial** data you **already had** from **other programs**. Copy/cut and paste is not restricted to the first time you upload data into MS-Access. It will be **quite frequent** that you want to upload new data doing copy/cut and paste from external programs (e.g., from Excel). It will also be quite frequent to copy/cut data from MS-Access and paste it into other programs (Excel, Word,...) for processing or reporting.

You may click:

- “*E.5.1 How do I copy/cut data from an MS-Access Table, Form or Query result?*”
- “*E.5.2 How do I paste data into MS-Access?*”
- “*E.5.3 How do I paste data copied from MS-Access into other applications?*”

### **E.5.1 How do I copy/cut data from an MS-Access Table, Form or Query result?**

MS-Access allows you to **copy/cut** data **from** a Table or Form and to **copy** data from a Query result.

Open the Table/Form/Query (from which you want to **copy** or **cut** data) in “**Datasheet View**” (click *B.4.1.3*).

You can **copy** or **cut** the **complete Table, Form** or **Query results, some contiguous rows, some contiguous columns**, one **cell** or one **rectangle of cells**. Notice I am calling **cell** to one field of one specific record.

**Select** the data that you want to **copy/cut** (click *B.5.2*). You now have the following two options:

- Right-click anywhere within the selected area and click on either “**Copy**” or “**Cut**” from the pop-up menu.
- Press “**Ctrl-c**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**c**” key) to **copy** the data.
- Press “**Ctrl-x**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**x**” key) to **cut** the data.

You can now go to the other application and **paste** the data that you have just copied/cut from MS-Access. You can also **paste** it into MS-Access, but, before pasting any data

into MS-Access, I suggest you click “[E.5.2 How do I paste data into MS-Access?](#)” to be aware of a few tricky issues.

### **E.5.2 How do I paste data into MS-Access?**

When pasting data into an MS-Access Table/Form there are four ways of pasting:

- **Paste into a field’s value**  
This is **pasting** copied **text** into the **string** that represents the value of a **field** while it is being edited.  
You may click “[E.5.2.1 How do I paste into a field’s value?](#)”.
- **Paste to edit one record.**  
This is **pasting** one (or more) fields over either the **new-record row** or over **one existing** record, in order to **edit one new** or **existing** record.  
You may click “[E.1 How do I edit one new or existing record?](#)”
- **Paste over a rectangle of fields**  
This is **pasting** a copied **rectangle of values** over a **rectangle of fields** (over **more than one** record) in order to **modify several existing** Table records.  
You may click “[E.5.2.2 How do I paste over a rectangle of fields?](#)”
- **Paste as new records.**  
This is **pasting** a copied **rectangle of values** (having **more than one** row) into the **new-record row**, in order to **enter several new** records into your Table.  
You may click “[E.5.2.3 How do I paste as new records?](#)”

You may also directly click:

- “[E.5.2.1 How do I paste into a field’s value?](#)”
- “[E.5.2.2 How do I paste over a rectangle of fields?](#)”
- “[E.5.2.3 How do I paste as new records?](#)”
- “[E.5.2.4 What are the fields and field order when pasting?](#)”

#### **E.5.2.1 How do I paste into a field’s value?**

**Open** the Table/Form (to which you want to paste) in “**Datasheet View**” (click [B.4.1.1](#)).

**Select** the **field’s value** (click [B.5.3](#)) where you want to paste.

**Move** (click [E.2.2](#)) the type-in cursor “|” to the place of the **string** representing the value where you want to **paste**.

You now **paste** in either of the following ways:

- Right-click inside the field and click “**Paste**” from the pop-up menu.
- Press “**Ctrl-v**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**v**” key).

If you are pasting a **string** and it does not work, you may click [L.4.1.2](#).

#### **E.5.2.2 How do I paste over a rectangle of fields?**

This subsection applies to **pasting** a **rectangle** of values over a **rectangle** of fields over **more than one** existing record. If you just paste over **one** existing record, click

*“E.1 How do I edit one new or existing record?”*.

**Open** the Table/Form (to which you want to paste) in **“Datasheet View”** (click *B.4.1.1*).

**Select** the **destination rectangle of fields** (click *B.5.2*) where you want to **paste** into. You can then either:

- Right-click within the **selected** rectangle of fields and click on **“Paste”** from the pop-up menu.
- Press **“Ctrl-v”** (i.e., press the **“Ctrl”** key, and without releasing it, press the **“v”** key).

If you **paste** without having previously **selected** the destination rectangle of **fields**, MS-Access will attempt to paste the whole copied rectangle of values into the currently selected **field’s value** (the field with its borders highlighted in **pink** color), which will most likely produce nonsense.

MS-Access will **start pasting** the **copied rectangle of values** at the **top-left** field of the **destination rectangle of fields**. It will then paste values **rightwards** and **downwards** from that **top-left field**. In case that the geometry (number of rows **and** number of columns) of the **copied** and the **destination** rectangles is **not** the same, the following two rules apply:

- Every **destination field** that does **not** have a matching **copied value** (i.e., a copied value in **the same** relative position) remains unchanged (there is no value to paste!).
- Every **copied value** that does **not** have a matching **destination field** (i.e., a destination field in the same relative position) is ignored (there is no field to paste into!).

MS-Access will **never modify** the records **blindly**, and rather, will **always** perform **error checks over each and every modified record**. If you want to know more about pasting errors, you may click *“L.4.4 How do I fix errors when pasting records into a Table/Form?”*.

**Pasting** over a **rectangle of fields** applies to pasting over the whole Table/Form, pasting over **one column** (one field across all records), pasting over a **column range** (several contiguous fields across all records) and pasting over a **row range** (several contiguous **existing** records). The only requirement is to paste over **more than one** existing record. If you want to know what pasting error you can get, you may click *“L.4.4 How do I fix errors when pasting records into a Table/Form?”*.

### **E.5.2.3 How do I paste as new records?**

This subsection applies to **pasting more than one** row of values into the **new-record row**. If you just paste **one** row of values, click *“E.1 How do I edit one new or existing record?”*.

**Go to** the **new-record row** (click *E.1.1*) and **select** the **whole new-record row** by clicking on the **star “\*”** icon on its leftmost side. The **whole row** should now have a **pink** border and a **light blue** background: this tells you that the **new-record row** has been properly selected. Notice that selecting the **whole new-record row** is different from only selecting **some** of its **fields**.

**Paste** the previously copied values (more than one row) in either of the following ways:

- Right-click within the **new-record row** and click on “**Paste**” from the pop-up menu.
- Press “**Ctrl-v**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**v**” key).

If the **number of columns** that you had previously copied (or cut) is **different** from the **number of columns** in the **Table/Form**, either of two following rules apply:

- If you copied **more** columns than the ones in the Table/Form, all the **additional** columns in the copied data will be **discarded**.
- If you copied **less** columns than the ones in the Table, **all the additional** columns in the Table/Form will be **set to Null** (if the error checking allows for it) in **all the newly pasted records**.

MS-Access will **never enter** the records **blindly**, and rather, will **always** perform **error checks over each and every record resulting from the paste operation**. If you want to know what pasting error you can get, you may click “*L.4.4 How do I fix errors when pasting records into a Table/Form?*”.

#### **E.5.2.4 What are the fields and field order when pasting?**

When doing copy/cut and paste, the **fields and field order** are exactly what you are seeing in “**Datasheet View**” in the Table/Form or Query result. Whatever fields or field order exists in “**Design View**” or in “**SQL View**”, is totally irrelevant: for the purpose of copy/cut and paste, what you see in “**Datasheet View**” is what you get.

#### **E.5.3 How do I paste data copied from MS-Access into other applications?**

When **pasting** data that you copied from a Table or Form or Query result (all in “**Datasheet View**”) you should be aware that the data may be **pasted differently** from what you copied. In the following subsections I explain some differences in content and formatting that may occur when you paste:

- “*E.5.3.1 How do I paste as plain text into Excel?*”
- “*E.5.3.2 How do I paste as formatted data into Excel?*”
- “*E.5.3.3 How do I paste as plain text into Word?*”
- “*E.5.3.4 How do I paste as formatted text into Word?*”
- “*E.5.3.5 What are the additional rows when pasting into Excel or Word?*”
- “*E.5.3.6 How do I paste into a plain-text processor?*”
- “*E.5.3.7 How do I paste back into MS-Access?*”

#### **E.5.3.1 How do I paste as plain text into Excel?**

Data is pasted with the font and formatting corresponding to the style existing in the Excel area where you paste. The data is pasted as the same rectangle of cells as it was copied from MS-Access, but, with **two additional rows**. The **leftmost cell of the first row** contains the Table/Form/Query **name** from which you copied the data. The **second row** of cells contains the **column headings** of the Table/Form or Query results from

which you copied the data. The data you copied/cut begins in the **third** row of cells.

### Pasting copied Short Text or *String* fields

Values of data/field type **Short Text** or *String* are pasted correctly. However, the pasted values may **appear to be different** in case the Table/Query/Form field was **showing a different** string from the **stored value**. If you want to know more about this, you may click “*L.7.1 Why text strings can be different from what is shown?*”.

### Pasting copied Yes/No or *Boolean* fields

Values of data/field type **Yes/No** or *Boolean* are pasted as follows, depending on the “**Format**” property of the field:

- “**Format=True/False**”: pasted as the corresponding Excel **Boolean values**. Excel will **display “TRUE”** or “**FALSE**”.
- “**Format=Yes/No**”: pasted **text strings** are “**Yes**” or “**No**”.
- “**Format=On/Off**”: pasted **text strings** are “**On**” or “**Off**”.

### Pasting copied Number, Currency, *Byte, Integer, Long, Single, Double* or *Currency* fields

**Very surprisingly**, pasting as plain-text into Excel preserves their data type and formatting, **but it will change some values!** Numeric **values** will be pasted **as you view them** in MS-Access. This means that number values that are **formatted** in MS-Access with “**n**” decimal digits, are pasted as **values** with “**n**” decimal digits!

Notice that this can cause **significant problems** because copy/cut and paste is **changing the values** of the numeric data. This is **really unusual**, because almost all programs handling number data **store** the values with full precision, independently of the **formatting** that you are using to **view** them.

### Pasting copied Date/Time or *Date* fields

When pasting values copied from MS-Access **Date/Time** or *Date* data/field types, a number of **anomalies** may happen. The cause of these anomalies is that Excel **can only** correctly handle dates between 1-March-1900 and 31-December-9999 (both included). For this reason, pasting **Date/Time** or *Date* values whose MS-Access format **only shows** the **time** will work well, but pasting values that have **date-only** or **date-and-time** formatting in MS-Access will experiment the following anomalies:

- **Date/Time** or *Date* values earlier than or equal to 31-December-1899:  
Excel will paste them as a **text string**, therefore destroying the processable value. The date and time **shown** in the text string will be correct.
- **Date/Time** or *Date* values between 1-January-1900 and 28-February-1900, both included:  
Excel will **paste a different numeric value** than the one in MS-Access, **but**, both the formatting and the value **shown** will be correct. This is so because the numeric value pasted is the one used by Excel, that in this case is different from the one in MS-Access.
- **Date/Time** or *Date* values later than or equal to 1-March-1900:  
Excel will paste the correct value, with correct formatting and the value shown will

be correct.

If you want to know more about the way **Date/Time** or *Date* values are stored/represented in MS-Access, you may click “D.4.5 What is the “Date/Time” field type?”.

### **E.5.3.2 How do I paste as formatted data into Excel?**

Data is pasted with the same font and formatting as displayed in MS-Access. The data is pasted in the same rectangle of cells as it was copied from MS-Access, but, with **one** (not **two**, as when pasting as plain text) **additional rows**. The **first row** of cells contains the **column headings** of the Table/Form or Query results from which you copied the data (pasted as text string) and will have gray background. Copied data starts in the **second** row of cells.

#### **Pasting copied Short Text or *String* fields**

The **values, data type and formatting** of **Short Text** or *String* is **preserved** when pasting.

However, the pasted **values** may **appear to be different** in case the Table/Form or Query results field was **showing** a **different** string from the **stored value**. If you want to know more about this, you may click “L.7.1 Why text strings can be different from what is shown?”.

#### **Pasting copied Yes/No or *Boolean* fields**

Values of data/field type **Yes/No** or *Boolean* are pasted as the corresponding Excel **Boolean values**. Excel will **display** “TRUE” or “FALSE”.

#### **Pasting copied Number, Currency, *Byte, Integer, Long, Single, Double* or *Currency* fields**

Their **values, data type and formatting** are **preserved** when pasting. Preserving formatting means that **the way you see** the values (e.g., thousands separator and decimal digits for a number) will be most similar as you were viewing them in MS-Access.

**Notice that, contrary to what happens when pasting as plain-text**, in this case the number **values are pasted correctly**. The value of numbers will be pasted as they were **stored** in MS-Access, independently of the way they were **viewed** in the MS-Access Table/Form or Query result.

#### **Pasting copied Date/Time or *Date* fields**

When pasting values copied from MS-Access **Date/Time** or *Date* data/field types, a number of **anomalies** will happen. The cause of these anomalies is that Excel **can only** correctly display dates between 1-March-1900 and 31-December-9999 (both included) and that **Excel** is not fully compatible with MS-Access formatting for **Date/Time** or *Date* values. For these reasons, pasting **Date/Time** or *Date* values in Excel as formatted will **correctly paste all the numeric values**, but the **formatting** and the values **shown** will experiment the following anomalies:

- **Date/Time** or *Date* values whose MS-Access format **shows** the time-part: Pasting these values will result in a **plain number** in Excel. This is, the MS-Access formatting will be lost.

- **Date/Time** or *Date* values whose MS-Access format **does not show** the time:
  - **Date/Time** or *Date* values earlier than 30-December-1899  
Excel will show them as “#####” because Excel will attempt to show them as dates, but Excel cannot show negative values as dates.
  - **Date/Time** or *Date* value 30-December-1899:  
Excel will **wrongly show** it as “0-jan-1900”.
  - **Date/Time** or *Date* values between 31-December-1899 and 27-February-1900:  
Excel will **wrongly show** them as 1 day more. For example, 31-December-1899 is shown as **1-January-1900** and 27-February-1900 is shown as “**28-February-1900**”.
  - **Date/Time** or *Date* value 28-February-1900:  
Excel will **wrongly show** it as “**29-February-1900**” (notice that 1900 is not a leap year).

If you want to know more about the way **Date/Time** or *Date* values are stored/represented in MS-Access, you may click “D.4.5 What is the “Date/Time” field type?”.

### **E.5.3.3 How do I paste as plain text into Word?**

Data is pasted with the font and formatting corresponding to the style existing in the Word **paragraph** where you paste it. The data is pasted as **one paragraph** of text for **each row** of copied data from MS-Access, but, with **one additional paragraph**. The **first paragraph** contains the **column headings** of the Table/Form or Query results from which you copied the data. The data you copied/cut begins in the **second paragraph**.

The text arising from different cells in each MS-Access row is delimited in Word with an intermediate **tab** character. The data from each cell (numbers, dates, text strings, ...) is pasted as text strings as was displayed in the MS-Access screen. For the case of **Yes/No** or *Boolean* values displayed in MS-Access as a **checkbox**, their pasted text depends on the value of the field’s “**Format**” property, as follows:

- “**Format=True/False**”: pasted **text strings** are “**True**” or “**False**”.
- “**Format=Yes/No**”: pasted **text strings** are “**Yes**” or “**No**”.
- “**Format=On/Off**”: pasted **text strings** are “**On**” or “**Off**”.

### **E.5.3.4 How do I paste as formatted text into Word?**

Data is pasted with the same font and formatting displayed in MS-Access. Data is pasted as a **Word table**. The **first row** of the table is a **single** table-wide cell that contains the Table/Form/Query **name** from which you copied the data. The **second row** of the table contains the **column headings** of the Table/Form or Query results from which you copied the data. The first two rows will have gray background. Copied data starts in the **third** row of the table. The data from each cell (numbers, dates, text strings, ...) is pasted as text strings as was displayed in the MS-Access screen. For the case of **Yes/No** or *Boolean* values displayed in MS-Access as a **checkbox**, their pasted text depends on the value of the field’s “**Format**” property, as follows:

- “**Format=True/False**”: pasted **text strings** are “**True**” or “**False**”.



- “Format=Yes/No”: pasted **text strings** are “Yes” or “No”.
- “Format=On/Off”: pasted **text strings** are “On” or “Off”.

### **E.5.3.5 What are the additional rows when pasting into Excel or Word?**

The different **additional rows inserted** when pasting are the following:

- **Pasting as plain text into Excel**  
Two additional rows are pasted. The **first** row has the Table/Form/Query **name** in the **leftmost** cell. The **second** row has the **column headings**.
- **Pasting as formatted text into Excel**  
**One** additional row is pasted, having the **column headings**. The cells in this additional row have gray background.
- **Pasting as plain text into Word**  
**One** additional paragraph is pasted, having the **column headings**.
- **Pasting as formatted text into Word**  
**Two** additional rows are pasted. The **first** row is a **single** table-wide **cell** having the Table/Form/Query **name**. The **second** row has the **column headings**. The cells in these two additional rows have gray background.

### **E.5.3.6 How do I paste into a plain-text processor?**

You can also paste into any plain-text processor, or any application that takes plain-text as input. The result will be the same as I explained in “*E.5.3.3 How do I paste as plain text into Word?*”, subject to the different behavior that each plain-text processor may have (e.g., splitting a paragraph into several independent lines, converting tabs into spaces, ...).

### **E.5.3.7 How do I paste back into MS-Access?**

Of course, you can also paste the data you just copied into an MS-Access Table: I explain how to paste into MS-Access in the next chapter *E.5.2*.

## **E.6 What checks are done when saving a field value, or entering or modifying a record?**

You may click:

- “*E.6.1 What checks are done when saving a field value?*”
- “*E.6.2 What checks are done when entering or modifying a record?*”

### **E.6.1 What checks are done when saving a field value?**

When you attempt to **save** (click *Part E*) a field value, MS-Access will always do the following **checks**:

- Check that the value **complies** the **field type** and **size**  
Click “*L.4.2.1 How do I fix an invalid value?*”.
- If the field is configured as “**Required=Yes**” (click *D.5.1.7*), check that its value is **non-Null**. Remind that **all Key fields** are **always** configured as “**Required=Yes**”.  
Click “*L.4.2.2 How do I fix trying to save Null in a “Required” field?*”.

- If it is a **Short Text** field configured as “**Allow Zero Length=No**” (click *D.5.2.1*), check that its value is **not** a string of zero length.  
Click “*L.4.2.3 How do I fix violating “Allow Zero Length=No”?*”.
- If **pasting** a text string, check that the length of the string is lesser or equal than the “**Field Size**” property of the **Short Text** field.  
Click “*L.4.2.4 How do I fix that I cannot paste a text string in full?*”.
- If you chose the value using a **drop-down menu** configured to only accept values from the menu (i.e., “**Limit to List=Yes**”), check that the value exists in the drop-down menu. Notice that this check **will not** be done when **pasting** values.  
Click “*L.4.2.5 How do I fix a value rejected because it is not in the list?*”.
- If the field has a **validation rule** (click *D.5.1.5*), check that its **Boolean** expression does not return **False**. Recall from *D.5.1.5* that if the **Boolean** expression returns **Null**, the field value is considered **valid**.  
Click “*L.4.2.6 How do I fix a value violating a field validation rule?*”.

If **one or more** of the above checks is **wrong**, MS-Access shows an error **message** and/or requests you to **fix** the field value. The way MS-Access does this depends on whether you are:

- **Editing** the **record under edition** or **pasting** over a rectangle of field values over **one** record (click *E.1*).
- **Pasting** several rows of values as new records or **pasting** over a rectangle of field values over **several** records (click *L.4.4*).
- **Searching** and **replacing** data in records (click *E.7.1*).
- **Cascade updating slave fields** (click *L.4.3.4*).
- **Inserting** or **updating** records with an SQL Query.

### **E.6.2 What checks are done when entering or modifying a record?**

When you attempt to **enter** (click *Part E*) a record, MS-Access will always do the following error checks over the record as a whole:

- If the Table has a **record validation rule** (click *D.8.1*), check that its **Boolean** expression over the record fields does not return **False**. Recall from *D.8.1* that if the **Boolean** expression returns **Null**, the record is considered **valid**.
- If one field, or a group of fields, are configured as the **primary Key** (click *D.6*) or have an associated **index without duplicate values** (click *C.8.3.2*), check that **value array** of that fields is **different from** the **value array** in every other record already in the Table.
- If one field, or a group of fields, are **slave fields** in a Relationship **with referential integrity** (click “*C.11 What is a Relationship?*”), check that their **value array** is the same as the one of their related **master fields** in **at least one record already existing** in the corresponding **master Table**.
- If one field, or a group of fields, are **master field(s)** in one (or more) Relationship(s) **with referential integrity** (click “*C.11 What is a Relationship?*”) with the option “**Cascade Update Related Fields**” (click *D.9.4*), check the correctness of **all** the

resulting **slave records** if updating the value(s) of the related **slave field(s)**. This means checking all the

If **one or more** of the above record error checks is **wrong**, MS-Access shows an error **message** and/or requests you to **fix** the record. The way MS-Access does this depends on whether you were entering:

- The **record under edition** (click *E.1.2*).
- **Records pasted** as new records or **records** modified by **pasting** over a rectangle of field values over **several** records (click *L.4.4*).

## E.7 How do I bulk-change my Table/Form's data?

This chapter also answers the question:

- **How I do I change erroneous values in a Table/Form field?**

You can do bulk-changes in your Table/Form's data in either of the following ways:

- “*E.7.1 How do I bulk-modify my data using the “Find/Replace” tool?*”  
Using the “**Find/Replace**” tool is best when the number of records to modify is small and the modification to perform is quite direct (i.e., it does not involve complex operations over the searched data).
- “*E.7.2 How do I bulk-change my data with an external application?*”  
Using an **external application** (e.g., Excel) is preferable when you are doing bulk changes that are **not repetitive**, and when the involved data is **not too large** (e.g., thousands of records).
- “*E.7.3 How do I bulk-change my data using data-changing Queries?*”  
Using **Queries that change Table data** is preferable when the data changing operation (e.g., remove records older than five years) is repeated periodically, or when the involved data is very large (e.g., tens of thousands of records, or more).



Notice also that bulk-changing your data is a **quite risky** operation, so **always** make a **backup** of the MS-Access file **before** doing the bulk changes. Click:


- “*E.7.4 Why should I backup my data before a bulk-change?*”

In spite of being risky, bulk-changing your data is an **extremely useful** operation. It is particularly convenient when you want to correct errors in your Table's data, like removing **Nulls**, zero-length strings or invisible strings, removing records with **duplicate** values or fixing records that do not comply with validation rules.

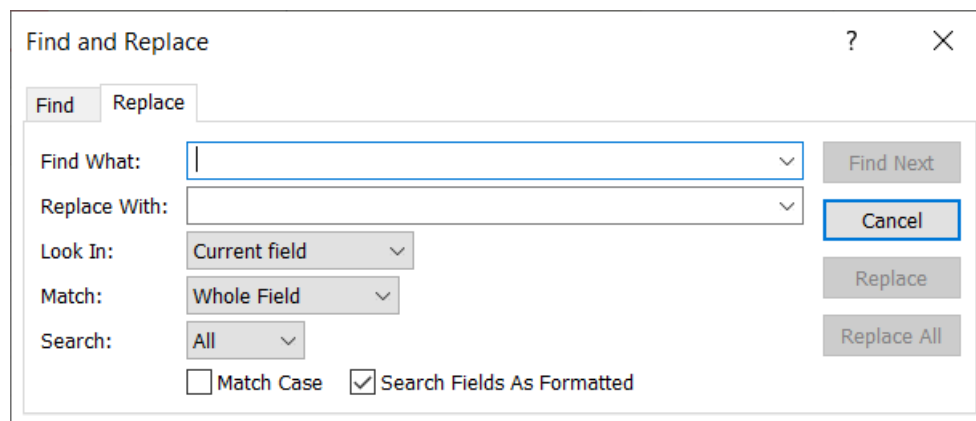
### E.7.1 How do I bulk-modify my data using the “Find/Replace” tool?

Show the “**Replace**” tab of the “**Find and Replace**” dialog box in either of the following ways:

- Click on “**Home**” from the Ribbon-bar and then on the **Replace** “” icon from the Ribbon.
- Click on the **Find/Replace** “” icon from the “**Quick Access Toolbar** (if you had previously added it) and then click on the “**Replace**” tab of the “**Find and Replace**” dialog box.

- Click on on “**Home**” from the Ribbon-bar and then click the **Find** “” icon. Finally, click on the “**Replace**” tab of the “**Find and Replace**” dialog box.

Regardless of how you showed it, you will see the “**Replace**” tab of the “**Find and Replace**” dialog box, as displayed in the next screenshot:



You can then type-in the searched string, the replace string, and set a few options using the drop-down menus and checkboxes. You can then either:

- Find and replace **manually**, match by match, clicking the “**Find Next**” and “**Replace**” buttons as required. This is very safe, but it is only suitable for a very low number of matches.
- Do a bulk replace by clicking the “**Replace All**” button.

### What errors can I get doing one by one “Replace”?

Each field value over which you do “**Replace**” will be checked for **field errors** (click *E.6.1*):

- If an **error** is encountered, the corresponding error message will be displayed. You have to close the error dialog box by either clicking on its “**OK**” button, or on its close “**X**” icon. Once the dialog box is closed, the erroneous **field’s value** will remain selected. MS-Access will not allow you to do anything until you either enter a correct field value or press the “**Esc**” key to **cancel** the value replacement of this field. Either if you **corrected** or **canceled**, you may continue with the replace operation.
- If **no error** is encountered, the field value is replaced and you may continue with the replace operation.

When the last replaced field of each record is attempted to be saves, the record as a whole is checked for **record errors** (click *E.6.2*):

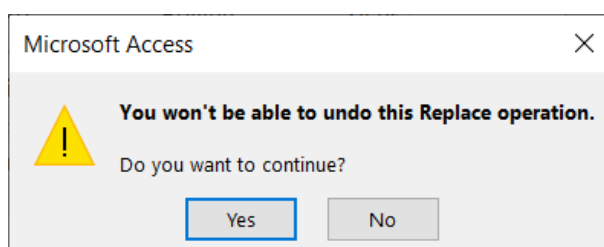
- If an **error** is encountered, the corresponding error message will be displayed. You have to close the error dialog box by either clicking on its “**OK**” button, or on its close “**X**” icon. Once the dialog box is closed, the **field’s value** of the last replaced field will remain selected. MS-Access will not allow you to do anything until you either enter a correct field value or press the “**Esc**” key to **cancel** the value replacement of this field. Notice that after you enter a correct field value, the record as a whole may still be erroneous. You may then correct other field values of the

record to make the record correct or press the “**Esc**” key to cancel the editing of this record. Either if you **corrected** or **cancel**, you may continue with the replace operation.

- If **no error** is encountered, the replaced record is **entered** and you may continue with the replace operation.

### What errors can I get doing “Replace All”?

If you do a “**Replace All**”, each field value that is replaced will be checked for field errors (click *E.6.1*) and for record errors (click *E.6.2*). For each different error that is encountered, the corresponding error message will be displayed. You have to close each displayed error dialog box by either clicking on its “**OK**” button, or on its close “**X**” icon. When finishing the replace, in case there was one or more records successfully modified, you will get the following dialog box:



If you click on “**No**”, the Replace operation will be **cancelled**, and **no record** will be modified.

If you click on “**Yes**”, the replaced records that successful passed **all** checks will be **modified**, while the replaced records that had one (or more) errors, will remain unmodified.

### E.7.2 How do I bulk-change my data with an external application?

You do this by **copying/cutting** the required **rectangle** of cells of Table data and **paste** it in an external application (typically Excel).

In the external application you use formulas (or whatever programming it offers) over the data to detect the values that you want to change (e.g., **Nulls**, zero-length strings or values that return **False** from the field validation rule). You should carefully design a formula or script that automatically produces the correct value for each record and/or field that you want to change.

Once you have produced the values or records that you want, you **copy** the corresponding rectangle of values from the external application and **paste** it MS-Access.

If you want to know more about doing copy/cut and paste with your data, you may click “*E.5 How do I copy/cut and paste data between MS-Access and other applications?*”.

### E.7.3 How do I bulk-change my data using data-changing Queries?

You do this by creating specific Queries that do the data changes that you want over your Tables. To do this you need to know how to code SQL Queries, and in particular, Queries that can change your Table’s data. If you want to know more about this, you may click “*F.13 How do I write a Query that changes my Table data?*”.

### E.7.4 Why should I backup my data before a bulk-change?

As a relevant overall advice, notice that a bulk-change operation involving **hundreds** or **thousands** of records is **inherently very risky**. Doing bulk-change operations you could inadvertently **destroy** or **corrupt** a **huge** number of records in your database. Fixing this later may be extremely difficult and time consuming. Therefore, my strong advice is that you do the following:

- **Backup** your database **right before** doing large bulk-change operations. This will allow you to go back to a correct state of the database in case there were problems with your bulk-change operations.
- Think **very carefully** what you are doing and **be very sure** that **all** the bulk-change operations you will be doing **are correct**.
- **Check quite thoroughly** the database data after the bulk-change, to make sure no problem has occurred, and that all database records and fields are correct. You should **at the very least** run your data check Query (click *K.6.11*).

### E.8 How do I upload my pre-existing data into my database?

When you create your database, you will need to populate its different Tables with data. Most frequently, you do not start from zero, entering new records about new events, and rather, you want to enter into the database a bulk of data you already have.

If you create a database of invoices, you will most likely want to enter the data of all your current customers and suppliers. You will also want to enter your current product list, and product list prices. You may want also to enter the current available units of each product, and you will likely want to enter also the invoices from the last months.

This means that you will want to **copy** the data from whatever program(s) and/or file(s) you currently have it and **paste** it into the **Tables** you have defined in MS-Access. You can also **import** data from other program(s) and/or file(s) into your Tables.

However, the format of the data (cell placement, intermixing with textual cells, rows, fields, data type...) that you **currently** have will for sure **not match** the format of your newly designed database Tables. You will therefore need to process your existing information in order to convert it to the format of the different database Tables. You can do this using the best tool for each case of existing data. You can use programs such as spreadsheets (Excel, ...), stream editors (sed, ...), math packages (R, ...), python scripts or other tools. Notice that you can also **use MS-Access** itself. You can create a migration database where you have Tables with the format of your **existing** information, and Tables with the format of your newly designed database. Then you write **specific Queries** to convert the data from the **current Tables** to the **new Tables**.

Regardless of how you will adapt your existing information, you will for sure end up doing substantial copy/cut and paste from other programs/files into your new database. You will use copy/cut and paste not only when you initially populate your database, but also all along its lifetime.

Copying/cutting and pasting text between plain text applications is trivial, but doing copy/cut and paste in MS-Access has a number of tricky issues. If you want to know how to do copy/cut and paste with some detail, and in particular, what errors can arise when doing it, you may click "*E.5 How do I copy/cut and paste data between MS-Access*"

*and other applications?”.*

As a **first** overall advice, notice that a **paste** operation into MS-Access involving **hundreds** or **thousands** of records is **inherently very dangerous**. Doing bulk paste operations you could inadvertently destroy or corrupt a **huge** number of records in your database. Fixing this later may be extremely difficult and time consuming. Therefore, it is strongly recommended to **both**:

- a) Backup your database before doing this type of bulk operations. This will allow you to go back to a correct state of the database in case you make a mistake with your bulk operations.
- b) Think **very carefully** what you are doing and **make very sure** that **all** the bulk operations you will be doing **are correct**.

## **E.9 Can I get inconsistent results out of my initial data in my database?**

Yes, you will usually get inconsistent (“wrong”) results when you apply Queries over your newly created database. This **does not** always mean that the database design or Query design is wrong. On my cases the problem is that when you populate initially your database, in particular if you upload diverse already existing data, it is very difficult to make all the data consistent timewise. The best you can get is partial consistency of the Queries over your oldest data, and it is **very important** that you understand well which Query results will be correct and which ones will not.


This explanation I just gave is really abstract, so I will try to clarify it with an example. Imagine that you upload in one shot the list of your current building projects, and also your past building projects up to some past moment in time. You also upload data on your expenses on another Table. A first consistency problem may be between expenses and projects. If you upload expenses corresponding to a certain number of past years, you will be missing all the project expenses of projects that started earlier than the first year of your data on expenses. You may think that this is solved by uploading all the expenses starting from the starting date of the earliest uploaded project. However, this will most likely cause wrong results in every Query providing data about yearly expenses, because the first year only has **part** of the total expenses. You may now think to upload all the missing expenses in the earliest year, but this is likely not possible because expenses must be linked to an existing project (click “[C.11 What is a Relationship?](#)”). One trick in such situations is to create “ad-hoc” objects to solve it. For example, you could create a false “Legacy” project, and bind to it all expenses bound to projects not uploaded in the database. Another solution is just to be aware that the results of some Queries over some of the oldest records uploaded in the database will be wrong.

The bottom line is that when you start working with the database you should decide what past data you will be uploading, and in what grade it is consistent. This will help you understand what Query results can be correctly expected over past data, and which ones not. Otherwise, you may be puzzled by some results that seem to indicate that the database is not working correctly, when it is actually working properly.

## **E.10 Why should I use “Compact and Repair Database”?**

Because as you use your database, its files get larger than is needed and may also get

corrupt. As you use your database, MS-Access adds to the files temporary Tables, indexes and other temporary data that make them larger. Also, this temporary data may be inconsistent with new data that has been added to your Tables, or with modifications you have done on your Tables. These inconsistencies may eventually make MS-Access return wrong results from your Queries.

Clicking on the **Compact and Repair Database**  icon from the “**Database Tools**” Ribbon makes MS-Access remove all the temporary data, and perform a check on consistency, fixing any possible cross-data errors.

My advice is you click on “**Compact and Repair Database**” quite frequently (e.g., every day, if you use the database a lot, or every few days if you use the database a little).

If you click on the **Compact and Repair Database**  icon, and you get the error message:

*“Could not find field 'Field\_name'.”*

this is most likely because you removed a field *'Field\_name'* that was used in a **Calculated** field and/or in the Table’s record validation rule. If you want to **fix** this, you may click *L.8.10*.

Notice that if you have a distributed database (click *K.3*), “**Compact and Repair Database**” works only over your **local** Tables and will have **no effect** on your **linked** Tables. Therefore, you will have to **run** it by opening your **backend** files and **cannot** do it **from** your **frontend** files.



## PART F. WRITING SQL QUERIES TO USE MY DATABASE

You can read this *Part F* in “**normal mode**”, by actually creating the example Tables and Queries that I propose, or, in “**lightning mode**”, by downloading the database and just running the examples while following my indications. For “**lightning mode**” you can download the MS-Access database with all the examples of Tables, Forms and Queries in this **Lightning Guide** from the link:

[https://lightningguide.net/Company\\_Database.zip](https://lightningguide.net/Company_Database.zip)

You may click:

- “*F.1 What is the Structured Query Language (SQL)?*”
- “*F.3 Why should I write and run Queries?*”
- “*F.4 What is an SQL operation and an SQL Query?*”
- “*F.5 How do I edit my SQL Queries with the plug-in “Access SQL Editor”?*”
- “*F.6 What are the SQL operators I use to write my Queries?*”
- “*F.7 What is a Select operation and how do I write it?*”
- “*F.8 What is a Join operation and how do I write it?*”
- “*F.9 What is a Union operation and how do I write it?*”
- “*F.10 What is a Transform operation and how do I write it?*”
- “*F.11 What are the SQL clauses, their expression’s elements and color codes?*”
- “*F.12 How do I add parameters (type-in variables) to my Queries?*”
- “*F.13 How do I write a Query that changes my Table data?*”
- “*F.14 How do I write and debug my SQL Queries?*”

### **F.1 What is the Structured Query Language (SQL)?**

The Structured Query Language (SQL) is an extremely widespread functional programming language to write database Queries. It is used in most database systems and is a widespread standard. There are a few syntax and semantics SQL variations from one database system to another, but a very large part of the syntax and semantics of the SQL they use will be the same. It is therefore a good investment to learn SQL, because it will allow you to work not only in MS-Access, but also in many other database systems.

The SQL language has operators and functions that can be combined to perform very complex operations over **record-lists** (click C.3.3), producing as a result a new **record-list**. SQL allows a very powerful and flexible way to select, merge, combine, split, and perform many other operations over **record-lists**. Along this *Part F* you may learn what are the different SQL operators and how to use them to write your Queries.

For those of you familiar with VBA, Java, C, C++, or any other **imperative** language, you will find that SQL is **very** different, and may look somehow weird. This is because SQL is a so-called **functional** language. Functional languages are not like imperative

languages in which you have a set of data structures that you define, a set of variables and an instruction flow that computes things. In functional languages you rather have values and expressions that compute other values out of a set of initial values. The main data type handled in an SQL operation is the **record-list**. An SQL **Query** is a **named** piece of SQL code that takes some **input record-lists**, some **other** possible inputs, and produces an **output record-lists** as a result.

## F.2 What version of SQL is this guide for?

The MS-Access SQL is largely, but not completely, compatible with the ANSI-92 SQL standard. MS-Access SQL has a few **additions** and a few **restrictions** over the ANSI standards.

Most relevant additions on my view are the extremely useful **Transform** operator, the very valuable “**PARAMETERS**” declaration and additional SQL aggregate functions like “**StDev ()**” and “**VarP ()**”.

Most relevant restrictions on my view are not allowing the “**DISTINCT**” clause in aggregate function references and not providing the “**LIMIT TO n ROWS**” clause.

You may check the official information from Microsoft about ANSI SQL and MS-Access SQL in the link:

[Comparison of Microsoft Access SQL and ANSI SQL](#)

Related with this, MS-Access has an option to align itself with either ANSI-89 or ANSI-92 standards. On my experience, the difference is negligible and my advice is to keep the MS-Access default setting, which is “ANSI-89”.

For these reasons, the SQL presented in this **Lightning Guide** is the one of MS-Access, under the MS-Access ANSI-89 option.

## F.3 Why should I write and run Queries?

Because each Query produces **one** of the many **complex data results** that you want to perform over the data in your database.

Running Queries is **the best part** of using a database. It is when you get the **fast results** you want from your database, even if they imply **very complex** calculations and data processing over your database.

To run the Queries, you need to first write them. You can write as many Queries as you need to get the complex results you want out of your database.

**Using** a database consists of **inputting** data records into the database Tables, **coding** Queries to exploit the data in the database, and **running** the Queries to obtain the desired summaries, statistics, reports, etc. out of your database.

## F.4 What is an SQL operation and an SQL Query?

You may click:

- “[F.4.1 What is an SQL operation?](#)”
- “[F.4.2 What is an SQL Query?](#)”

- “F.4.3 Why should I write my Queries in SQL?”
- “F.4.4 What SQL Query editor should I use?”
- “F.4.5 How do I create a Query?”
- “F.4.6 How do I edit my SQL Queries without a plug-in Query editor?”

#### F.4.1 What is an SQL operation?

An **SQL operation** is a **formula** composed of an **SQL operator**, **input record-lists** and other operands that produces **one single output record-list**. Each of its **input record-lists** can be a **Table name**, a **Query name** or **another SQL operation**.

A SQL operation is conceptually the same as a conventional expression. The SQL operation makes its computations over record-lists and uses SQL operators to build the operations over other operations, while an expression makes its computation over scalar values of given data type(s) (e.g., integer numbers, fractional numbers, dates, ...) using conventional operators (e.g., “+”, “-”, “\*”, “/”, ...) and functions (e.g., “Log()”, “Iif()”, ...). In spite of being conceptually the same, the syntax and semantics of an SQL operation and the ones of a conventional value expression are very different.

An SQL operation (unless it contains non-deterministic functions) is **deterministic** and does not have memory. This means that if you compute several times a given SQL operation **over the same input record-lists**, you **always get the same output** record-list.

#### F.4.2 What is an SQL Query?

A Query is a **named** and **self-contained** piece of SQL code that is stored in MS-Access.

Once a Query is defined and stored in MS-Access, you may **run** (click *B.4.1.9*) the Query whenever you want, to get the result that the Query will produce. **Query names must be unique** in each MS-Access database, to avoid ambiguity when invoking a Query.

The SQL operations in the Query code may take as its **input** record-lists Table names and/or other Query names (called “**auxiliary** Queries”) that exist in the database. Therefore, the SQL code of a Query may contain references to **Table names** and/or to other **Query names** of the database.

Most Queries are “**consulting**” Queries. You can think of a **consulting** Query as an SQL operation taking as its **input** record-lists database Tables and/or other Queries, and after processing them it produces **one output** record-list. A consulting Query will **not modify** the data in the database Tables: it just **reads** data from the Tables and processes it to produce **one output** record-list.

Some more advanced Queries (“**modifying**” Queries) may **insert**, **delete** or **update** the records in the database Tables. If you want to know more about this, you may click “F.13 How do I write a Query that changes my Table data?”.

#### F.4.3 Why should I write my Queries in SQL?

MS-Access allows you to create Queries with a proprietary semi-graphical user interface. I think this proprietary interface is cumbersome, very limited in its expressing power, and prone to errors. I will not cover this proprietary user interface and will only

explain how to create Queries using the SQL language. Learning SQL will allow you to use other database systems in addition to MS-Access, because many database systems use SQL. The SQL language in other systems will have some minor variations in respect to the one used in MS-Access, but most of the logic and fundamental ideas will be the same.

Writing your Queries in SQL can cause some minor issues in MS-Access: it may sometimes report a warning, indicating that the Query was not correct, because it cannot convert it to its internal semi-graphical interface. This is particularly likely if you open the Query in “**Design View**”. This is not a relevant issue and has an easy fix.

#### **F.4.4 What SQL Query editor should I use?**


MS-Access allows you to edit your Queries in SQL. However, it **does not** allow to write **comments** in the SQL code, and it **destroys the code formatting**. These are **very severe** problems and in practice you **should not** write your SQL Queries with the MS-Access text interface.

You should therefore buy a **plug-in** SQL editor for MS-Access. This will allow you to write your Queries **directly** into MS-Access, with **comments** and proper **formatting**. I have been using the plug-in “**Access SQL Editor**” for years and I like it **very much**. This editor provides the basic functionalities (regexp search and replace, keyword highlight, comment/uncomment and parentheses matching) that cover **most** of what you need, it is very simple to use, and its **price is really low!!** If you buy it, it will be your **second<sup>23</sup>** best investment ever. If you want to know more about the plug-in “**Access SQL Editor**”, you may click “*F.5 How do I edit my SQL Queries with the plug-in “Access SQL Editor”?*”.

While you buy a plug-in SQL editor, you may write your Queries in a standalone editor, and then copy them into a Query opened in “**SQL View**” (click *B.9*). If you want to know more, you may click “*F.4.6 How do I edit my SQL Queries without a plug-in Query editor?*”.

#### **F.4.5 How do I create a Query?**


##### **If you are not using the plug-in “Access SQL Editor”**

Click on the “**Create**” from the “**Ribbon-bar**”, and then click on the **Query Design**  icon. This will create a Query with a default Query name (“**Query1**” or something similar) displayed in a new “**Query pane**” in “**Design View**”.

In order to **edit** the SQL code of the Query, you have to change its **view-type** to “**SQL View**” (click *B.4.1.4*). You can now **edit** the SQL code of your Query (click *F.4.6*). If you want to **rename** your Query, click *B.4.1.8*. If you want to **save** your Query, click *B.4.1.6*. If you want to **run** it, click *B.4.1.9*. Finally, if you want to **close** it, click *B.4.1.7*.

##### **If you are using the plug-in “Access SQL Editor”**

Open the plug-in “**Access SQL Editor**” (click *F.5.1.2*).

Click on the **New**  “**New**” icon (click *F.5.4.8*) from the “**Access SQL Editor**” toolbar.

---

<sup>23</sup> Your **first** best investment ever is this Lightning guide 😊 .

This **creates** a **new** Query and creates a new “Query pane” (click *F.5.6.1*) for it **inside** the **Access SQL Editor**” (click *F.5.2*). You may then **edit** (click *F.5.5*), **rename**, **save**, **run** or **close** (click *F.5.3*) your Query using the corresponding commands from the “**Access SQL Editor**”.

#### **F.4.6 How do I edit my SQL Queries without a plug-in Query editor?**

You can edit your Queries in “**SQL View**” but (as of the writing of this book) the editing capacities in “**SQL View**” are extremely poor. Microsoft announced that it would add the Monaco SQL editor to MS-Access along the second quarter of 2021 to provide good editing capacities.

Unless the Monaco editor is already added, my strong advice is that you install the plug-in “**Access SQL Editor**” into MS-Access (click *F.5*). If for some reason you do not want to install the “**Access SQL Editor**”, you may write your Queries in some standalone (i.e., not integrated in MS-Access) SQL editor application and **keep** them **stored outside** of MS-Access. Each time you modify a Query code, you open the MS-Access Query in SQL mode (click *B.9*), and copy/paste the Query code from your external editor into MS-Access. Notice that this makes the process of writing new Queries and debugging them quite painful, as **every modification** requires a copy/paste operation, and this is also prone to causing errors. This is why I advise you to install the “**Access SQL Editor**”.

If you want to **edit** an **existing** Query in “**SQL View**”, open the Query (click *B.4.1.3*) and then change its view-type to “**SQL View**” (click *B.4.1.4*). You may then directly edit the Query or copy/paste the complete SQL code from the standalone SQL/text editor you are using into MS-Access. Once you have edited the Query code, save the Query (click *B.4.1.6*) and you may then run it (click *B.4.1.9*). Once you have finished editing the query, you may close it (click *B.4.1.7*).

#### **F.5 How do I edit my SQL Queries with the plug-in “Access SQL Editor”?**

The description of the “**Access SQL Editor**” in this chapter is for its version “1.1.53.0”. If you have a newer version, notice that it will likely contain some improvements and minor differences from what is described here.

You may click:

- “*F.5.1 How do I start with the plug-in “Access SQL Editor”?*”
- “*F.5.2 What is the user interface of the plug-in “Access SQL Editor”?*”
- “*F.5.3 What are the toolbar commands in the “Access SQL Editor”?*”
- “*F.5.4 How do I manage the Queries/Tables with the “Access SQL Editor”?*”
- “*F.5.5 How do I edit a Query in its query pane in the “Access SQL Editor”?*”
- “*F.5.6 How do I configure the layout of the “Access SQL Editor”?*”

### F.5.1 How do I start with the plug-in “Access SQL Editor”?

You may click:

- “F.5.1.1 How do I get and install the plug-in “Access SQL Editor”?”
- “F.5.1.2 How do I open the plug-in “Access SQL Editor”?”
- “F.5.1.3 What options should I set in the “Access SQL Editor”?”





#### F.5.1.1 How do I get and install the plug-in “Access SQL Editor”?

You get the plug-in “**Access SQL Editor**” from the company “**Field Effect**” on-line at [fieldeffect.info](http://fieldeffect.info). You can start with a free trial and if you like it buy it later.

You install the “**Access SQL Editor**” by running its set-up file, and it will then be **embedded** into MS-Access.

#### F.5.1.2 How do I open the plug-in “Access SQL Editor”?

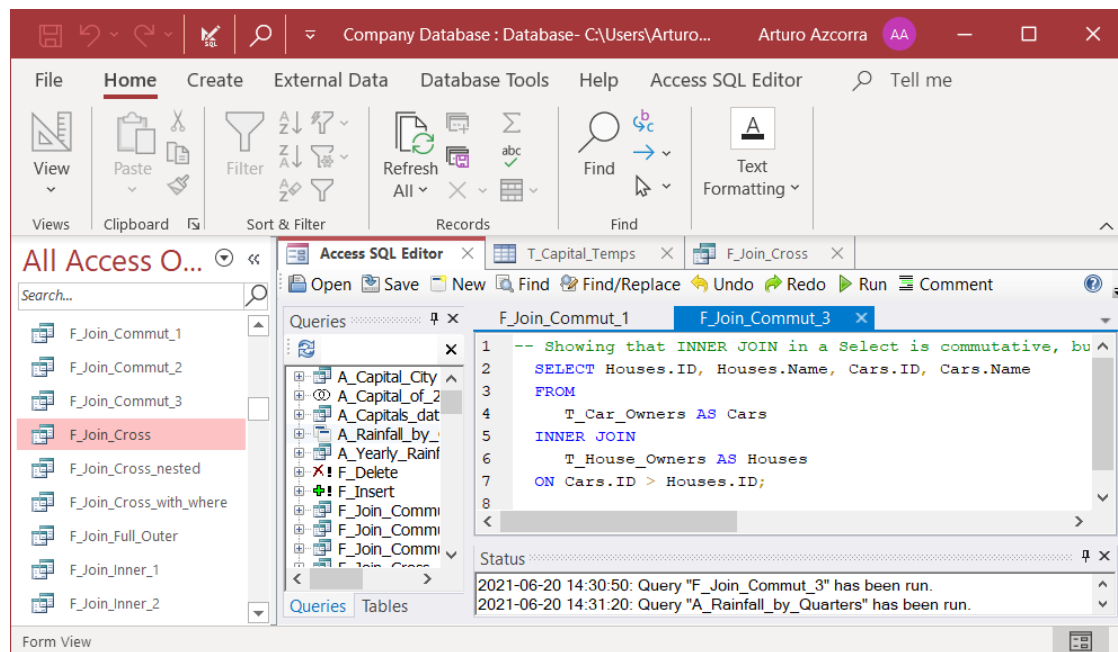
You **open** the “**Access SQL Editor**” in either of the following ways:

- If you **did add** the **Access SQL Editor**  icon to the “**Quick Access Toolbar**” (click *B.1.4*), just click on the  icon.
- If you **did not add** the **Access SQL Editor**  icon to the “**Quick Access Toolbar**”, click on “**Access SQL Editor**” from the “**Ribbon-bar**”. This will show a Ribbon that only contains the **Access SQL Editor**  icon: click on it.

Either way will **open** the “**Access SQL Editor**” in an MS-Access “**Object pane**” (click *B.2.10*) displayed in the MS-Access “**Object Area**” (click *B.2.8*). The tab heading of this “**Object pane**” is (surprise!) “**Access SQL Editor**”. The next screenshot shows the



“Access SQL Editor” in its specific “Access SQL Editor” pane:



In this screenshot you may see the “Access SQL Editor” pane currently displayed in the MS-Access “Object Area”, and the **tabs** of two additional MS-Access “Object panes”: “T\_Capital\_Temps” and “F\_Join\_Cross”, that are opened but not displayed. **Within** the “Access SQL Editor” pane you may see several of its objects (you may click [F.5.2](#) for an explanation of its user interface).

### F.5.1.3 What options should I set in the “Access SQL Editor”?

You click on its “Tools” icon and then click on “Editor Options...” from the pop-up menu. I suggest you **tick all** the options, except “Show query results in built-in grid (may be slow)” and “Ctrl+Q Toggles Comments”. You then click on “OK”, and close and open the “Access SQL Editor” for the changes to take place. I advise you **do not tick** the option “Show query results in built-in grid (may be slow)” because it has worse performance, the layout of the Query results may differ from the one of MS-Access, it can only show the results of one Query at a time, and on some rare cases it may cause crashes.

You click on the “Tools” icon and then place the mouse over “Windows”. You then **tick** “Queries”, “Tables” and “Status”, and leave “Results” **unticked**. You have to click on the “Tools” icon again for each **tick/untick** that you want to do. After you have configured this, the editor will show one “Queries” pane with the list of **Queries**, one “Tables” pane with the list of **Tables** and one “Status” pane listing a log of Query diagnostics.

You click on the “View” icon and then **untick** “Word Wrap”. This will show your SQL code as such, without word wrapping. If a line of code is wider than the viewing area, you will have to do horizontal scroll to view it. I think this is the best approach to view your code properly formatted.

If at any moment you are using a small screen, it may be useful to **tick** “Word Wrap”, but in the usual situation with a large screen I think it better to have it **unticked**.

### F.5.2 What is the user interface of the plug-in “Access SQL Editor”?

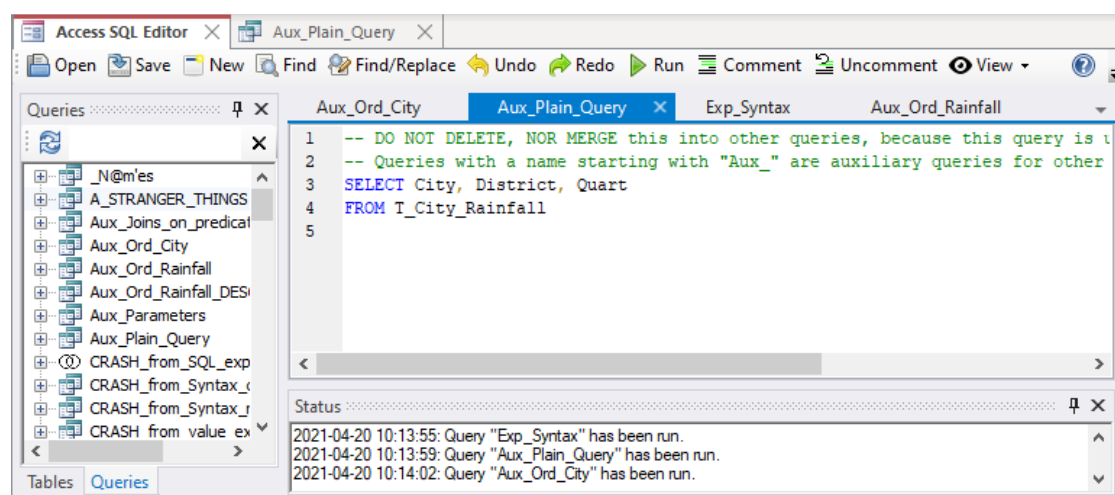
When you run it, the “**Access SQL Editor**” is shown in one MS-Access “**Object pane**”, in the MS-Access “**Object Area**”. This is, the “**Access SQL Editor**” is shown as if it were one more MS-Access object type, like an MS-Access Query, Table or Form.

The “**Access SQL Editor**” pane has a **toolbar** (click *F.5.3*) on top, and **below** the toolbar there is an **object area** that contains **pane-spaces** (click *F.5.6.6*) and/or **sub-windows** (click *F.5.6.7*). The **sub-windows** can **also** be contained in **self-standing windows** (click *F.5.6.8*). **Each pane-space** or **sub-window** hosts one, or more, object panes from the “**Access SQL Editor**”. You can configure the **layout** of these different elements as best suits you (click *F.5.6* or *F.5.6.11*).

The **object panes** of the “**Access SQL Editor**” are:

- **Several query panes**  
Each query pane shows the **SQL code** of **one** Query (click *F.5.6.1*).
- **One “Queries” pane**  
It lists the **names** of **all** the database **Queries** (click *F.5.6.2*).
- **One “Tables” pane**  
It lists the **names** of **all** the database **Tables** (click *F.5.6.3*).
- **One “Status” pane**  
It lists the **results** of your **commands** and **operations** (click *F.5.6.4*).
- **One “Results” pane**  
It shows the **results** of running a Query inside the “**Access SQL Editor**” (click *F.5.6.5*).

As an example, the following screenshot shows my preferred layout for the “**Access SQL Editor**” pane. You can see inside its object area one **sub-window** (left), hosting the “**Tables**” pane and the “**Queries**” pane, another **sub-window** (bottom-right) hosting the “**Status**” pane and a **pane-space** (top-right) hosting **four** query panes:



Do not mistake the “**Queries**” pane with a **query** pane. The “**Queries**” pane is just one specific pane that lists the **names** of **all** the Queries in your database. The query pane(s) are **one or more** object panes, **each** of which shows the **SQL code** of **one** specific Query. To clarify this difference, I write “**Queries**” pane using quotes and a different



font, while I write **query pane** without quotes and with the usual font.

Finally, I want to point out that you should **not mistake** the following elements:

- The **specific object area** of the “**Access SQL Editor**” (that is described in this chapter *F.5*) with the usual MS-Access “Object Area” (click *B.2.8*).
- The **specific object panes** of the “**Access SQL Editor**” (that are described in this chapter *F.5*) with the usual MS-Access “Object panes” (click *B.2.10*).
- The **specific query panes** of the “**Access SQL Editor**” (that are described in this chapter *F.5*) with the usual MS-Access “Query panes” that show a Query in different view-types (click *B.5*, *B.7* or *B.9*).









Notice that to facilitate distinguishing them, I write the **specific** elements of the “**Access SQL Editor**” without quotes and in lower case, while I write the **usual** elements of MS-Access between quotes and capitalized.





### F.5.3 What are the toolbar commands in the “Access SQL Editor”?




While you are working with the “**Access SQL Editor**” you will use the command icons in its **toolbar** (located at the **top** of the “**Access SQL Editor**” pane, right below its tab), and you will **not** usually use the commands of MS-Access. The following screenshot shows the “**Access SQL Editor**” toolbar:




The commands in the “**Access SQL Editor**” **toolbar** are, left to right:

- **Open** “ Open”  
**Opens** a Query/Table: click *F.5.4.1*.
- **Save** “ Save”  
**Saves** the Query in the **current** query pane: click *F.5.4.6*.
- **New** “ New”  
**Creates** a new Query: click *F.5.4.8*.
- **Find** “ Find”  
Opens the “**Find and Replace**” dialog box in its “**Find**” tab: click *F.5.5.2*.  
Do not mistake the “**Find and Replace**” dialog box of the “**Access SQL Editor**” with the one of MS-Access (click *B.5.5* and *E.7.1*).
- **Find/Replace** “ Find/Replace”  
Opens the “**Find and Replace**” dialog box in its “**Replace**” tab: click *F.5.5.2*.  
Do not mistake the “**Find and Replace**” dialog box of the “**Access SQL Editor**” with the one of MS-Access (click *B.5.5* and *E.7.1*).
- **Undo** “ Undo”  
**Undoes** the **last** editing operation that you performed: click *F.5.5.2*.
- **Redo** “ Redo”  
**Redoes** the **last** editing operation that you **undid**: click *F.5.5.2*.
- **Run** “ Run”  
**Runs** the Query: click *F.5.4.5*.

- **Comment** “ Comment”  
**Comments** the selected lines of code: click *F.5.5.1*.
- **Uncomment** “ Uncomment”  
**Uncomments** the selected lines of code: click *F.5.5.1*.
- **View** “ View ▾”  
This allows to set/unset **word wrapping** for showing the SQL code: click *F.5.1.3*.
- **Tools** “ Tools ▾”  
Shows some “**Access SQL Editor**” options (click *F.5.1.3*), and a couple **other tools**.  
The tools to convert tabs to spaces and vice versa can be quite useful.

Finally, notice that **each sub-window** (click *F.5.6.7*) that is **inside** a **self-standing window** (click *F.5.6.8*) **also** has a **toolbar**. This **toolbar** includes **the same** commands as the **main toolbar** (as explained above) **except** “ Open”, “ New” and “ Tools ▾”.

If the **width** of the “**Access SQL Editor**” pane, or of the **sub-window**, is not enough to show all the toolbar commands, the ones that **do not fit** will **not** be shown. In this case, the pop-up menu “” icon will be **shown** on the **rightmost** side of the toolbar. If you click on this icon, a pop-up menu will show all the toolbar commands that cannot be shown for lack of width, and you will be able to click on them.

#### **F.5.4 How do I manage the Queries/Tables with the “Access SQL Editor”?**

You may click:

- “*F.5.4.1 How do I open a Query/Table from the “Access SQL Editor”?*”
- “*F.5.4.2 How do I select a Query/Table name in the “Access SQL Editor”?*”
- “*F.5.4.3 How do I show a query pane in the “Access SQL Editor”?*”
- “*F.5.4.4 How do I select a query pane in the “Access SQL Editor”?*”
- “*F.5.4.5 How do I run a Query from the “Access SQL Editor”?*”
- “*F.5.4.6 How do I save a Query from the “Access SQL Editor”?*”
- “*F.5.4.7 How do I close a query pane in the “Access SQL Editor”?*”
- “*F.5.4.8 How do I create a new Query from the “Access SQL Editor”?*”
- “*F.5.4.9 How do I rename a Query from the “Access SQL Editor”?*”
- “*F.5.4.10 How do I copy a Query from the “Access SQL Editor”?*”
- “*F.5.4.11 How do I delete a Query from the “Access SQL Editor”?*”

##### **F.5.4.1 How do I open a Query/Table from the “Access SQL Editor”?**


You **open** a Query/Table (from the “**Access SQL Editor**”) in either of the following ways:

- Double-click<sup>24</sup> on the Query or Table **name** from the “**Queries**” pane (click *F.5.6.2*),

---

<sup>24</sup> I am assuming that you **set** the recommended option “Double-clicking queries opens them instead of executing”.

or “**Tables**” pane (click *F.5.6.3*), respectively.

- Right-click on the Query or Table **name** from the “**Queries**” pane (click *F.5.6.2*), or “**Tables**” pane (click *F.5.6.3*), respectively, and click on “**Open**” from the pop-up menu.
- **Select** the Query or Table **name(s)** (click *F.5.4.2*) that you want. You then either:
  - Click on the “ **Open**” icon from the “**Access SQL Editor**” toolbar.
  - Right-click on one of the selected Query or Table **name(s)** and click on “**Open**” from the pop-up menu.
  - Press the “**Enter**” key.

If you **select several** Query or Table **names** you can **open all** of them in **one shot**.

Regardless of how you open a **Table**, it will be opened in its own MS-Access “Table pane” (click *B.2.10*) **outside** the “**Access SQL Editor**”, in the MS-Access “Object Area” (click *B.2.8*).

Regardless of how you open a **Query**, it will be opened in its own query pane (click *F.5.6.1*) **within** the “**Access SQL Editor**”. The query pane will be **labeled** with the **name** of the Query, and it will be **placed** as follows:

- If there is an **empty** area in the “**Access SQL Editor**” object area, the new query pane will be displayed in a **pane-space** in the formerly **empty** area.
- If there is **no empty** area, and the “**Queries**” pane is **inside** a **sub-window**, the new query pane will be displayed in a **pane-space** as follows.
  - If a **pane-space** that is hosting query panes **already exists**, the new query pane will be displayed in that already existing **pane-space**.
  - Otherwise, a **new pane-space** will be created (typically in the top-middle area of the object area), and the new query pane will be displayed in that newly created **pane-space**.
- If there is **no empty** area, and the “**Queries**” pane is **inside** a **pane-space**, the new query pane will be displayed in this same **pane-space**. This means that it will be displayed **instead of** the “**Queries**” pane. This on my view is quite uncomfortable, so my advice is you **always** place the “**Queries**” pane in a **sub-window**, and **never** in a **pane-space**.

Although a newly opened query pane is **always** displayed in a **pane-space** (click *F.5.6.6*), you can **move** it (click *F.5.6.9*) into a **sub-window** (click *F.5.6.7*), either within the “**Access SQL Editor**” object area or within a **self-standing window** (click *F.5.6.8*).

If you **open** a Query that is **already opened**, this will **select** its query pane (click *F.5.4.4*). Notice that you **cannot** have the same Query opened in two query panes.

If you **open** a Query and it is directly marked as containing **unsaved changes** (i.e., it has an asterisk to the left of its name), you may click *L.8.4*.

#### **F.5.4.2 How do I select a Query/Table name in the “Access SQL Editor”?**

You **select one** Query/Table **name** (in the “**Access SQL Editor**”) by clicking on it from

the “**Queries**” pane (click *F.5.6.2*), or “**Tables**” pane (click *F.5.6.3*), respectively.

You select **several individual** Query/Table **names** (in the “**Access SQL Editor**”) by clicking on **one** of them, and then doing **Ctrl+click** on **each** of the **other ones** that you want to select. Notice that if you do **Ctrl+click** on a selected Query/Table name, it will become unselected.

You select **a range of** Query/Table **names** (in the “**Access SQL Editor**”) by clicking on **one end** of the range, and then doing **Shift+click** on the **other end** of the range.

Notice you can first select a range of Query/Table **names** doing **Shift-click** and **then** select (or unselect) **several individual** ones doing **Ctrl+click**.



The **selected** Query/Table **name(s)** become **highlighted** with **white text over blue background**.

#### **F.5.4.3 How do I show a query pane in the “**Access SQL Editor**”?**

The way to **show** one query pane (in the “**Access SQL Editor**”), making it the **currently shown one**, depends on where it is hosted, as follows:

- If the query pane is hosted in a **pane-space** (click *F.5.6.6*), you **show** the query pane by clicking on its **tab** (at its top). The **currently shown** query pane has its **tab** highlighted with either a **white text over blue background** or a **black text over gray background**.
- If the query pane is hosted in a **sub-window** (click *F.5.6.7*), or inside a **self-standing window** (click *F.5.6.8*), you **show** the query pane by clicking on its **tab** (at its bottom). The **currently shown** query pane will have its **tab** highlighted with a **blue text**.

#### **F.5.4.4 How do I select a query pane in the “**Access SQL Editor**”?**

The “ Save” and “ Run” commands from the **toolbar** apply to the **currently selected** query pane. It is therefore important to know **which** is the **currently selected** query pane and also, how to **select** a given query pane to make it the **currently selected one**.

The way to **select** one query pane depends on where is it hosted, as follows:

- If the query pane is hosted in a **pane-space** (click *F.5.6.6*), you **select** the query pane that you want by clicking on its **tab** (at its top). This will also make it the currently shown query pane in the **pane-space**. Notice that in a **pane-space** the currently selected query pane is **not highlighted**, which makes it more difficult to know which one is it (read further below).
- If the query pane is hosted in a **sub-window** (click *F.5.6.7*), and the sub-window contains **more than one** query pane, you **select** the query pane that you want by clicking on its **tab** (at its bottom) of the query pane. This will also make it the currently shown query pane in the **sub-window**. The **top frame** of the sub-window hosting the currently selected query pane is highlighted with a **white name over blue background**.

You may also **select** a query pane by **opening** its associated Query (click *F.5.4.1*):

- If the Query was **already** opened, this will **select** its query pane.
- If the Query was **not** opened, this will open the Query and make its query pane **the**

**currently selected one.** If you open **several** Queries in one shot, the currently selected query pane will be the **last one** in being opened (the shown one).

In the “**Access SQL Editor**” object area or in each **self-standing window**, the **currently selected** query pane is identified as follows:

- If there is a **sub-window** with its top frame highlighted with a **white name over blue background**, then the currently selected query pane is the currently shown one in that **sub-window**.
- If there is **no sub-window** with its top frame highlighted with a **white name over blue background**, then the currently selected query pane is the one highlighted with a **white name over blue background** in a **pane-space**.

#### F.5.4.5 How do I run a Query from the “Access SQL Editor”?

You **run** a Query (from the “**Access SQL Editor**”) in either of the following ways:

- **Select** the query pane (click *F.5.4.4*) that you want and click the “▶ **Run**” icon from the toolbar (click *F.5.3*).
- Right-click on the Query **name** from the “**Queries**” pane (click *F.5.6.2*) and click on “**Run**” from the pop-up menu.
- **Select** the Query **name(s)** (click *F.5.4.2*) that you want from the “**Queries**” pane. You then either:
  - Click the “▶ **Run**” icon from the toolbar (click *F.5.3*).
  - Right-click on any of the selected Query **names** and click on “**Run**” from the pop-up menu.

If you **select several** Query **names** you can **run all** of them in **one shot**.

The corresponding Query **results** will be shown in an MS-Access “Query pane” in “**Datasheet View**”<sup>25</sup> in the MS-Access “Object Area”, this is, **outside** the “**Access SQL Editor**”.

You will **frequently run** a Query from the “**Access SQL Editor**” while you are writing or debugging it.


When you have been working on another MS-Access “Object pane”, or on another application, and afterwards you click on the “▶ **Run**” icon to run a Query in a pane that **appears** to be selected, many times it does not work. In this case, you just need to click on the query pane to activate the focus and click again on “▶ **Run**”.

You may **also** run the Query from the MS-Access “**Navigation Pane**” (click *B.4.1.9*). However, my advice is that you **hide** the “**Navigation Pane**” while you are using the “**Access SQL Editor**”, to prevent some possible interferences between both of them. Therefore, my advice is that you **do not run** the Queries from the “**Navigation Pane**” while you are using the “**Access SQL Editor**”.

---

<sup>25</sup> I am assuming that you **did not** set the non-advisable option “Show query results in built-in grid (may be slow)”.

#### F.5.4.6 How do I save a Query from the “Access SQL Editor”?

You **save** the **editing changes** on the SQL code (from the “**Access SQL Editor**”) by selecting the query pane (click *F.5.4.4*) and clicking on the “ Save” icon from the **toolbar**.

You can **also save** the SQL code by **closing** the query pane (click *F.5.4.7*), and then clicking on “**Yes**” in the confirmation window, although this is somehow risky.

**Before** actually saving the SQL code of a query pane, the “**Access SQL Editor**” will do a **syntax check**. If the Query code contains some types of syntax errors, MS-Access will **not** save the unsaved editing changes and will show a **syntax error** message. If you want advice on fixing this, you may click:

- “*J.8 How do I fix a syntax error that prevents saving a Query?”*”

Finally, notice that clicking on “**Save As...**” from the pop-up menu over the Query **name** will **not** save the editing changes of its query pane. Clicking on “**Save As...**” creates a **copy** of the Query (click *F.5.4.10*), **without** the unsaved changes, and its query pane will remain **the same**.

#### F.5.4.7 How do I close a query pane in the “Access SQL Editor”?

The way to **close** a query pane (in the “**Access SQL Editor**”) depends on where the query pane is **hosted**, as follows:

a) If the query pane is **hosted** in a **pane-space** (click *F.5.6.6*), you **close** it in either of the following ways:

- Click on the close “**X**” icon placed on the **tab** (at the top) of the query pane.
- Right-click on the **tab** (at the top) of the query pane and click on “**Close**” from the pop-up menu.
- Right-click on the **tab** (at the top) of the query pane and click on “**Close all BUT this**”.

Doing this will **close all the other** query panes, from **all pane-spaces**, **all sub-windows** and **all self-standing windows**.

b) If the query pane is **hosted** in a **sub-window** (click *F.5.6.7*), you **close** it in either of the following ways:

- Click on the “**▼**” icon placed on the top **sub-window frame**, or rather, **right-click** anywhere on the top **sub-window frame**, or else, **right-click** on the **tab** (at the bottom) of the query pane. Then click on “**Close**” from the pop-up menu. Notice that right-clicking on the **tab** only applies if there is **more than one** query pane in the sub-window (otherwise there will be **no tabs**).

Doing this will **only close** the currently displayed query pane.

- Click on the “**X**” icon on the right of the **top frame** of the **sub-window**.

Doing this will **only close** the currently displayed query pane.

- Click on the “**▼**” icon placed on the top **sub-window frame**, or rather, **right-click** anywhere on the top **sub-window frame**, or else, right-click on the **tab** (at the bottom) of the query pane. Then click on “**Close all BUT this**” from the pop-up



menu. Notice that right-clicking on the **tab** only applies if there is **more than one** query pane in the sub- window (otherwise there will be **no tabs**).

Doing this will **close all the other** query panes, from **all pane-spaces, all sub-windows and all self-standing windows**.

- If the **sub-window** is **inside** a **self-standing window**, click on the “X” icon on the right of the **top frame** of the **self-standing window**.

Doing this will **close the self-standing window itself**, and therefore, it will **also close all its sub-windows, and all their query panes**.

If you close a query pane that has **unsaved changes** in its SQL code, you will get a confirmation dialog-box asking if you want to save it, showing the buttons: “Yes”, “No” and “Cancel”:

- Clicking “Yes” **saves** the SQL code **and closes** the query pane. If you get a syntax error that prevents saving the SQL code, you may click *J.8*.
- Clicking “No” closes the query pane **without saving** the unsaved changes in the SQL code.
- Clicking “Cancel” **cancel**s the closing of the query pane.

Do not mistake closing a specific query pane from the “Access SQL Editor, with closing one of the usual MS-Access “Query panes” (click *B.4.1.7*).

Tables are **not** opened within the “Access SQL Editor”, and therefore you **close** them from the MS-Access “Navigation Pane” in the usual way (click *B.4.1.7*).

#### **F.5.4.8 How do I create a new Query from the “Access SQL Editor”?**

You **run** a Query (from the “Access SQL Editor”) by clicking on the “New” icon from the “Access SQL Editor” toolbar. This will **create** a **new** Query with the name “New\_nn” where “nn” is a double-digit integer number.

The new Query will be opened in its own query pane inside the “Access SQL Editor” and will be placed in the same way as when opening an existing Query (click *F.5.4.1*).

To give the name you want to the new Query click *F.5.4.9*.

#### **F.5.4.9 How do I rename a Query from the “Access SQL Editor”?**

The way to **rename** a Query (from the “Access SQL Editor”) depends on where it query pane is hosted, as follows:

- For any hosting type, right-click on the Query name from the “Queries” pane and click on “Rename” from the pop-up menu. This works even if the Query is not opened.
- If the query pane is **hosted** in a **pane-space** (click *F.5.6.6*), right-click on its **tab** (at the top) and clicking on “Rename” from the pop-up menu.
- If the query pane is **hosted** in a **sub-window** (click *F.5.6.7*), **click** on the “▼” icon placed on the top **sub-window frame**, or rather, **right-click** anywhere on the top **sub-window frame**, or else, **right-click** on its **tab**. Then click on “Rename” from the pop-up menu. Notice that right-clicking on the **tab** only applies if there is **more**

**than one** query pane in the **sub-window** (otherwise there will be **no tabs**). Notice also that if the **sub-window** is the only one within a **self-standing window** (click *F.5.6.8*), the “▼” icon is not shown.

Regardless of which way you clicked on “**Rename**”, you get a dialogue-box where you can type-in the new name that you want for the Query. When you are done typing-in, click on the “**OK**” button. If you rather want to cancel the renaming, click on the “**Cancel**” button.

#### **F.5.4.10 How do I copy a Query from the “Access SQL Editor”?**

You **copy** a Query (from the “**Access SQL Editor**”) by right-clicking on the Query **name** from the “**Queries**” pane and clicking on “**Save As...**” from the pop-up menu. You then type-in the name you want for the copy of the Query.

Notice that if the Query **being copied** (the query **name** where you did “**Save As...**”) was being edited in a query pane, and it had **unsaved changes**, the unsaved changes will **not** be included in the Query **copy**: the SQL code of the Query **copy** will be the one of the **last saved** version of the Query **being copied**.

#### **F.5.4.11 How do I delete a Query from the “Access SQL Editor”?**

You **delete** a Query (from the “**Access SQL Editor**”) in either of the following ways:

- Right-click on the Query **name** from the “**Queries**” pane (click *F.5.6.2*) and click on “**Delete**” from the pop-up menu.
- **Select** the Query **name(s)** (click *F.5.4.2*) that you want. You then either:
  - Right-click on one of the selected Query **name(s)** and click on “**Delete**” from the pop-up menu.
  - Press the “**Supr**” key.

If you **select several** Query **names** you can **delete** all of them in **one shot**.

Regardless of how you **delete** the Query(s), you will get a message-box informing you that Query deletion **cannot be undone** and asking for confirmation: you may click on “**Yes**” to proceed deleting the Query(s) or click on “**No**” to cancel.

### **F.5.5 How do I edit a Query in its query pane in the “Access SQL Editor”?**

You **open** the Query (click *F.5.4.1*) and you then **edit** its SQL code in its query pane. You **edit** its SQL code like in a conventional text editor but having also a few specific functionalities. If you want to know more, you may click:

- “*F.5.5.1 What are the specific editing functionalities of a query pane from the “Access SQL Editor”?*”
- “*F.5.5.2 What are the conventional editing functionalities of a query pane from the “Access SQL Editor”?*”

If you see an asterisk “\*” to the left of the Query **name** (in the query pane **tab** and/or in the **sub-window** top frame), this means that the query pane contains **unsaved** modifications of its SQL code.



### F.5.5.1 What are the specific editing functionalities of a query pane from the “Access SQL Editor”?

I describe below the **specific** SQL editing functionalities.

#### **Parentheses matching**

When you place the editing cursor after a parenthesis character (either “(“ or “)”), the editor will color in **red** **both** that parenthesis character **and** its **matching** parentheses (either “)” or “(”, respectively). This is **extremely useful** to write and debug your expressions and your other SQL code, saving you **a lot** of time and frustration arising from fixing parentheses errors.

#### **Syntax coloring**

In the SQL code of the query pane the keywords are colored in **blue**, the function names in **pink**, the operators, parentheses and commas in **brown**, the number constants in **orange**, the comments in **green** and the names in **black**. This coloring highly improves the readability of your SQL code. Notice that this coloring has **nothing to do** with the color codes that I use along this **Lightning Guide** to highlight different parts of the SQL code (click *F.11.2*).


#### **Line numbering**

Each SQL line of code is numbered on the left side, so you can know what part of the SQL Query code you are currently viewing.

#### **Indent selected text**


Pressing the “**Tab**” key indents all the lines in the selected text. This is useful for formatting your SQL code.

#### **Comment<sup>26</sup> selected text**


If you **click** the “ Comment” icon, for **each** of the **selected lines of text** in the **currently selected** query pane, it will be **commented**.

**Commenting** a line means **adding** two consecutive hyphens “**--**” **right before** the first non-blank character of the line. This is the same as **turning** the complete line into an SQL **comment**.

Pressing “**Ctrl-q**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**q**” key) also **comments all** the selected lines of text in the currently selected query pane (even if you set the non-recommended option “**Ctl+Q toggles comments**”).

Notice that if you have selected **whole** lines of text (e.g., by doing drag-and-drop over the left margin), pressing “**Ctrl-q**” will **also** comment the line of text **right after** the selection, which is quite annoying. This **does not** happen when you click the “ Comment” icon.

#### **Uncomment selected text**


If you **click** the “ Uncomment” icon, this **uncomments** **once** **all** the **selected lines of text** in the **currently selected** query pane.

**Uncommenting a line** **once** means **removing two** consecutive hyphens “**--**” when

---

<sup>26</sup> I am assuming that you **did not set** the option “**Ctl+Q toggles comments**”, following my advice.



they are the **first non-blank** characters of the line. This is the same as **turning** all the line text after the two removed hyphens (until the next two consecutive hyphens) from an SQL comment into valid SQL code.

Notice that if you have selected **whole** lines of text (e.g., by doing drag-and-drop over the left margin), the “ Uncomment” command will **also** uncomment the line of text **right after** the selection, which is quite annoying.

### F.5.5.2 What are the conventional editing functionalities of a query pane from the “Access SQL Editor”?

I describe below the **conventional** editing functionalities.

#### Find and Replace

Click on the “ Find” or “ Find/Replace” icons from the **toolbar**. This opens the “**Find/Replace**” dialog box in its “**Find**” tab or “**Replace**” tab, respectively.

The “**Find/Replace**” tool of the “**Access SQL Editor**” is particularly useful because it supports **regular expressions** (that the MS-Access find-and-replace does not support). If you are familiar with regular expressions, you will really like this. If you are not, I suggest you learn, because regular expression find-and-replace is **extremely** useful when writing, debugging and updating your SQL code. When doing find and replace using regular expressions, I suggest you **tick** the “**Find/Replace**” option “**Multiline**” (it appears on the “**Find/Replace**” dialogue-box) so it works over multiple lines. In case you want to do find and replace **only** on the **selected** text, then you also have to **tick** the option “**Search Selection**”.

The regular expression version used is the one of “.NET”. You can check the information about this version in the link [.NET Regular expressions](#).

As a final remark, it would be **extremely** useful if the plug-in “**Access SQL Editor**” would allow you to do find and replace over **all the Queries**, for example, to change the name of a user-defined VBA function **in all your SQL code** in one shot. Unfortunately, as of the date of release of this book it still does not have this functionality.

Do not mistake the “**Find/Replace**” dialog box of the “**Access SQL Editor**” with the “**Find and Replace**” tool of MS-Access (click *B.5.5* and *E.7.1*). Notice that the MS-Access “**Find and Replace**” tool will **not** work in the “**Access SQL Editor**” pane. If you try to use it, MS-Access will show a message indicating that it is not possible to do find and replace.

#### Select text

You **select text** in the query pane in the usual way: drag and drop with the mouse or click plus Shift-click. Selected text will be highlighted with **white text over blue background**.

#### Select all

You **select all** the text in the query pane (i.e., **all** the SQL code of the Query) in either of the following ways:

- Right-click inside the query pane and click on “**Select All**” from the pop-up menu.
- Press “**Ctrl-a**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**a**”

key) while the mouse is inside the query pane.

### **Move/Delete/Copy/Cut the selected text**

Selected text can be **moved** by doing drag and drop.

It can be **deleted** by pressing the “**Supr**” key or the “**Del**” key.

It can be **copied** by pressing “**Ctrl-c**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**c**” key).

It can be **cut** by pressing “**Ctrl-x**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**x**” key).

You can also **do the last three actions** by right-clicking inside the query pane, and then clicking on “**Delete**”, “**Copy**” or “**Cut**” (respectively) from the pop-up menu.


### **Paste text**

You can **paste** previously copied/cut text (from the editor or from any other Windows application) in either of the following ways:

- Right-click inside the query pane, and then click on “**Paste**” from the pop-up menu.
- Press “**Ctrl-v**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**v**” key).


### **Undo**

You can **undo** your last editing operation in either of the following ways:

- Clicking on the undo “ Undo” icon from the editor **toolbar**.
- Right-click inside the query pane, and then click on “**Undo**” from the pop-up menu.
- Press “**Ctrl-z**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**z**” key).

### **Redo**

You can **redo** the editing operation that you **undid** in either of the following ways:

- Click on the redo “ Redo” icon from the editor **toolbar**.
- Right-click inside the query pane, and then click on “**Redo**” from the pop-up menu.
- Press “**Ctrl-y**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**y**” key).

## **F.5.6 How do I configure the layout of the “Access SQL Editor”?**

You may click:

- “F.5.6.1 What are the query panes in the “Access SQL Editor”?”
- “F.5.6.2 What is the “Queries” pane in the “Access SQL Editor”?”
- “F.5.6.3 What is the “Tables” pane in the “Access SQL Editor”?”
- “F.5.6.4 What is the “Status” pane in the “Access SQL Editor”?”
- “F.5.6.5 What is the “Results” pane in the “Access SQL Editor”?”
- “F.5.6.6 What is a pane-space in the “Access SQL Editor”?”

- “F.5.6.7 What is a sub-window in the “Access SQL Editor”?”
- “F.5.6.8 What is a self-standing window of the “Access SQL Editor”?”
- “F.5.6.9 How do I move an object pane, a sub-window or a self-standing window of the “Access SQL Editor”?”
- “F.5.6.10 How do I convert between an object pane, a pane-space, a sub-window, or a self-standing window in the “Access SQL Editor”?”
- “F.5.6.11 What can be a good layout for the “Access SQL Editor”?”

### F.5.6.1 What are the query panes in the “Access SQL Editor”?

The purpose of **each query pane** of the “**Access SQL Editor**” is **viewing and editing** (click F.5.5) the SQL code of **one** Query.


**Each** query pane shows the **SQL code** of **one** Query that you **opened** for editing. **Each** query pane is labeled with the **name** of the Query whose SQL code it contains. If you see an asterisk “\*” to the left of the Query **name** (in the query pane **tab** and/or in the **sub-window** top frame), this means that the query pane contains **unsaved** modifications of its SQL code.


A query pane may be hosted in a **pane-space** (click F.5.6.6) or in a **sub-window** (click F.5.6.7).

Do not mistake the specific query panes of the “**Access SQL Editor**” described along this chapter F.5 with the usual MS-Access “Query panes” (click B.2.10).

### F.5.6.2 What is the “Queries” pane in the “Access SQL Editor”?






The “**Queries**” pane lists the **names** of **all** the database **Queries**, in alphabetical order. Notice that it **also** lists the Query **names** of Queries that are **hidden** in the MS-Access “**Navigation Pane**”. It is labeled “**Queries**”.

You **unhide** the “**Queries**” pane by clicking on the toolbar “ Tools ▾” icon, placing the mouse over “**Windows**”, and then **ticking** “**Queries**”.

You **hide** the “**Queries**” pane by either closing it “**X**” icon from its top frame or by clicking on the “ Tools ▾” icon, placing the mouse over “**Windows**”, and then **unticking** “**Queries**”.

On the leftmost side of each Query **name** there is a “**+**” icon. If you click on it, the Query’s **field names** will be listed below the Query **name**.

Each Query **name** is prefixed by an **icon** indicating the **type** of Query, as follows:


-  icon: **Select** Query
-  icon: **Union** Query
-  icon: **Insert** Query
-  icon: **Delete** Query
-  icon: **Transform** or **Update** Query

If you right-click on a Query **name**, a **pop-up menu** will be shown with the following self-explicative actions: “**Open**”, “**Run**”, “**Rename**”, “**Save As...**”, “**Delete**” and

“**Properties**”. Notice that the name of the “**Save As...**” action is quite misleading, because it **does not** save the Query being edited with a different name. It rather **copies** the Query as is currently **stored** in the editor (i.e., as you last saved it) with the new name that you provide after having clicked in “**Save As...**” (click *F.5.4.10*).

If you want to **open, edit**, or do other operations on your Queries, you may click:


- “*F.5.4 How do I manage the Queries/Tables with the “Access SQL Editor”?*”
- “*F.5.5 How do I edit a Query in its query pane in the “Access SQL Editor”?*”


If you click on the refresh “” icon at the top of the “**Queries**” pane, the list of Query **names** will be refreshed. Refreshing may be required if you did changes over the Queries from the “**Access SQL Editor**” and/or from MS-Access. Doing changes (e.g., deleting, creating, modifying, renaming, ...) over your Queries from MS-Access while you are also working over them from the “**Access SQL Editor**” is **absolutely non-advisable**, because it can create a number of collisions and problems.

If you type-in a **string** in the filter box “ ×” at the top of the “**Queries**” pane, **only** the Query **names containing that string** will be **shown**. This is very useful when you have lots of Queries. If you want to show back **all** the Query **names**, you can either delete the string or click on the clear filter “**X**” icon on the right of the filter box.

### **F.5.6.3 What is the “Tables” pane in the “Access SQL Editor”?**

It is labeled “**Tables**”. It shows the **names** of **all** the database **Tables**, in alphabetical order. Notice that it **also** shows the Table **names** of Tables that are **hidden** in the MS-Access “**Navigation Pane**”. It also shows the MS-Access **system** Tables with names starting with “**MSys**”: do not be puzzled by them and do not open them, just let them stay.

You **open** the “**Tables**” pane by clicking on the toolbar “ **Tools** ▾” icon, placing the mouse over “**Windows**”, and then **ticking** “**Tables**”.


You **close** the “**Tables**” pane by either clicking on the close “**X**” icon from its top-right or by clicking on the “ **Tools** ▾” icon, placing the mouse over “**Windows**”, and then **unticking** “**Tables**”.

On the leftmost side of each Table **name** there is a “**+**” icon. If you click on it, the Table’s **field names** will be listed below the Table **name**.

If you right-click on a Table **name**, a **pop-up menu** will be shown with just one choice: “**Open**”.

If you want to **open** Tables or **select** Table names, you may click:

- “*F.5.4.1 How do I open a Query/Table from the “Access SQL Editor”?*”
- “*F.5.4.2 How do I select a Query/Table name in the “Access SQL Editor”?*”

If you click on the refresh “” icon at the top of the “**Tables**” pane, the list of Table **names** will be refreshed. Refreshing may be required if you did changes over the Tables from the “**Access SQL Editor**” and/or from MS-Access.

If you type-in a **string** in the filter box “ ×” at the top of the “**Tables**” pane, **only** the Table **names containing that string** will be **shown**. This is very useful

when you have lots of Tables. If you want to show back **all** the Table **names**, you can either delete the string or click on the clear filter “X” icon in the filter box.

#### F.5.6.4 What is the “Status” pane in the “Access SQL Editor”?

It shows the **results** of your **commands** and **operations**, and most important, it shows the **error messages** when you **run** a Query from the “Access SQL Editor”. Error messages are shown in **red color** font (you can configure the color in the options).

#### F.5.6.5 What is the “Results” pane in the “Access SQL Editor”?

In case that you checked the **non-advisable** option “Show query results in built-in grid (may be slow)”, the “Results” pane shows the **result** of running a Query **inside** the “Access SQL Editor” instead of showing it in its own MS-Access “Query pane”. If you want to know why is it not advisable to check this option, you may click *F.5.1.3*.

#### F.5.6.6 What is a pane-space in the “Access SQL Editor”?

A **pane-space** from the “Access SQL Editor” contains one or more object panes and is used to display **one** of them in a flexible and configurable way.

Each **pane-space** has at its **top** a **list of tabs instead of** the thick top frame that **sub-windows** and **self-standing windows** have. The name of **each** object pane hosted **inside** the **pane-space** is shown as the **label of one** of the **tabs**. If the **pane-space** is too narrow to show all the tabs, the rightmost tabs that do not fit will **not** be shown. When this happens, the “▼” icon (on the right side of the **tabs**) **changes to** “⌵” to indicate that not all **tabs** are being shown (see below what it is for).

If the viewing area is too small to show all the contents of the currently displayed object, a horizontal and/or vertical scrollbar will be shown.

The **tab** of the **currently displayed** object pane is **highlighted with dark blue background** and a close “X” icon is shown on the rightmost side of its tab (see the screenshot below).

When the mouse is over any **tab** that is **not** the currently displayed one, the **tab** is **highlighted with medium-blue background** and a close “X” icon is shown on the rightmost side of the **tab**.

In the next screenshot (a zoom from the screenshot in *F.5.6.7*), you may see the Query **tabs** of a **pane-space**, which are “Aux\_Ord\_City”, “Aux\_Plain\_Query”, “Exp\_Syntax” and “Aux\_Ord\_Rainfall”.



You can **display one** of the object panes that are inside the **pane-space** by **selecting** it. You **select** an object pane by **clicking** on its **tab** (at the top of the **pane-space**). You can **also** select an object pane by **clicking** on the “▼” or “⌵” icon (see above) and then **clicking** on the name of the object pane that you want from the pop-up menu.

If you click on the close “X” icon placed on the right side of a **tab**, the corresponding object pane will be **closed**, and **another** object pane from the **pane-space** will be displayed. If the **pane-space** only contained the closed object pane, then the **pane-space** will also be **closed**.

When the **only** object pane in a **pane-space** is **closed** or **moved**, the former space of the



**pane-space** in the object area is either left empty or used to enlarge a neighboring **pane-space**.

You can change the **relative size** of **neighboring sub-windows** and/or **pane-spaces** by doing click-and-drag (i.e., pressing the right mouse button, moving the mouse, and then releasing) on the **separating frame** between them.

Finally, you can **move** any of the **object panes hosted** in the **pane-space outside** of it by doing drag-and-drop on the **object pane's tab**. If you want to know more about this, you may click "*F.5.6.9 How do I move an object pane, a sub-window or a self-standing window of the 'Access SQL Editor'?*".

### **F.5.6.7 What is a sub-window in the 'Access SQL Editor'?**

A **sub-window** from the "**Access SQL Editor**" contains one or more object panes and is used to display **one** of them in a flexible and configurable way.

Each **sub-window** has a **thick top frame**, like **self-standing windows** do, and unlike **pane-spaces** (that have instead a **list of tabs**). This thick top frame shows on its **left** side the **name** of the object pane that is **currently displayed** in the **sub-window**. The thick top frame also has on its **rightmost** side a **pin icon** ("📌" or "📌") and a **close "X"** icon (see below their functionality).

While a **sub-window** is displaying a **query pane**, a pop-up menu "▼" icon will be shown on the **left** side of its thick top frame (on the right side of the pin icon).

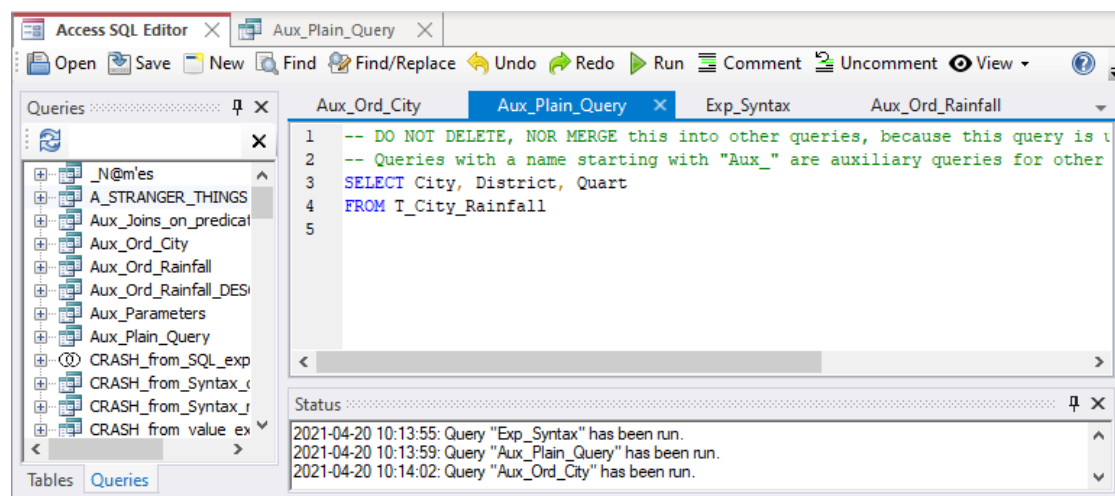
If a **sub-window** contains more than one object pane, the **names** of the object panes that it contains are shown as **tabs** at the **bottom** of the **sub-window**. In case there are tabs, the **name** of the currently displayed object pane is **highlighted with blue text** in its corresponding **tab**. If the **sub-window** is too narrow to show all the **tabs**, some of the **tab names** will be shortened to some initial part of the name followed by "...".

If the viewing area is too small to show all the contents of the currently displayed object, a horizontal and/or vertical scrollbar will be shown.

For the **specific case** of **sub-windows** that are **inside** a **self-standing window** (click *F.5.6.8*), they have **two** differences:

- Each of them has its own **toolbar**. This **toolbar** has the **same commands** as the main "**Access SQL Editor**" **toolbar**, **except** "📁 Open", "📄 New" and "⚙️ Tools ▼". If you want to know more about the **toolbar**, you may click *F.5.3*.
- They **do not have** the pin icon ("📌" or "📌"), because they do not have the autohide feature.

The next screenshot shows the “**Access SQL Editor**” pane, where its **object area** contains **one pane-space** (top-right) (click *F.5.6.6*) and **two sub-windows** (left and bottom-right) (click *F.5.6.7*):



This screenshot shows several features that I presented above:

- The bottom-right **sub-window** only has one object pane (the “**Status**” pane), and therefore it does **not** have **tabs**. The name of the currently displayed object pane (which is “**Status**”) is displayed on the left side of its thick top frame. Since the currently displayed object is not a query pane, the **sub-window** top frame does not have the pop-up menu “**▼**” icon.
- The left **sub-window** has two object panes, and therefore it **has two tabs** “**Tables** **Queries**” placed at its bottom. The name of the currently displayed object pane (which is “**Queries**”) is displayed on the left side of its thick top frame. You can also see that its name is highlighted with blue text in its **tab**. Since the currently displayed object is not a query pane, the **sub-window** top frame does not have the pop-up menu “**▼**” icon.
- You can see the **pin** “**📌**” icon and the **close** “**X**” icon on the right side of the thick top frame of both **sub-windows**.

You can **display one** of the object panes that are inside the **sub-window** by **selecting** it. You **select** an object pane by **clicking** on its **tab** (at the bottom of the **sub-window**).

If you click on the **close** “**X**” icon on the right side of the top frame, the **currently displayed** object pane inside **the sub-window** will be **closed**, and **another** object pane from the **sub-window** will be **displayed**. If the sub-window only contained the closed object pane, then the **sub-window** will **also** be **closed**.

When a **sub-window** is **closed** or **moved**, its **former** space in the object area is **occupied** by **enlarging** a neighboring **sub-window** or **pane-space**.

If you click on the pin icon (“**📌**” or “**📄**”) on the right side of the top frame, **sub-window autohide** is toggled **on** and **off**. If the pin icon is shown as “**📌**”, the sub-window is permanently displayed (autohide is **off**). If it is shown as “**📄**”, the **sub-window** will be automatically **hidden** (autohide is **on**) when you **select** an object pane from outside of the **sub-window**. My advice is you **do not use autohide** and leave all **sub-windows** permanently displayed.



You can change the **relative size** of **neighboring sub-windows** and/or **pane-spaces** by doing click-and-drag (i.e., pressing the right mouse button, moving the mouse, and then releasing) on the **separating frame** between them.

Finally, you can **move** a **sub-window** by doing drag-and-drop on its thick top frame. You can also **move** any of the **object panes** hosted in the **sub-window** outside of it by doing drag-and-drop on the **object pane's tab**. If you want to know more about this, you may click "[F.5.6.9 How do I move an object pane, a sub-window or a self-standing window of the "Access SQL Editor"?](#)".

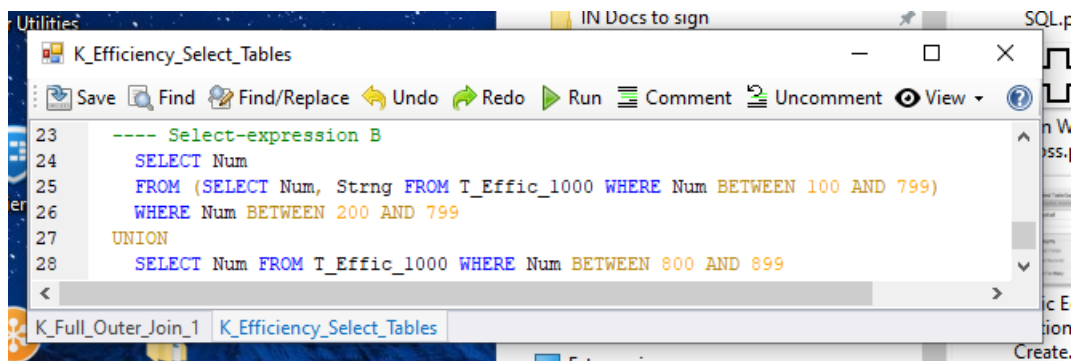
### F.5.6.8 What is a self-standing window of the "Access SQL Editor"?

A **self-standing window** from the "**Access SQL Editor**" is an application Window (as an Excel or Word Window) that contains one or more **sub-windows** (click [F.5.6.7](#)) and is used to display them in a flexible and configurable way.

Each **self-standing window** has a **thick top frame**, like **sub-windows** do, and unlike **pane-spaces** (that have instead a **list of tabs**). This thick top frame has on its **rightmost** side the usual minimize "–", maximize "□" and close "X" icons of Microsoft Windows.

If a **self-standing window** only contains **one sub-window**, the usual top thick frame of the **sub-window** will be removed, and the name of the currently displayed object pane will be shown on the right side of the top thick frame of the **self-standing window**.

The next screenshot shows a **self-standing window** that contains only **one sub-window**:



This screenshot shows several features that I presented above:

- The **self-standing window** has on the right side of its thick top frame the usual minimize, maximize and close Windows icons.
- Since there is **only one** enclosed sub-window, the **self-standing window** displays on the left side of its thick top frame "K\_Efficiency\_Select\_Tables" as the name of the currently displayed object pane.
- The enclosed **sub-window** has its own **toolbar**.
- The enclosed **sub-window** has two object panes, and therefore it **has two tabs** placed at its bottom. The name of the currently displayed object pane is highlighted with blue text in its **tab**.

If you click on the **close** "X" icon on the right of the top frame of the **self-standing window**, it will be **closed**, and therefore, **all** its **sub-windows** and **all** their object panes will **also** be closed. Clicking on the minimize "–" icon has no effect.

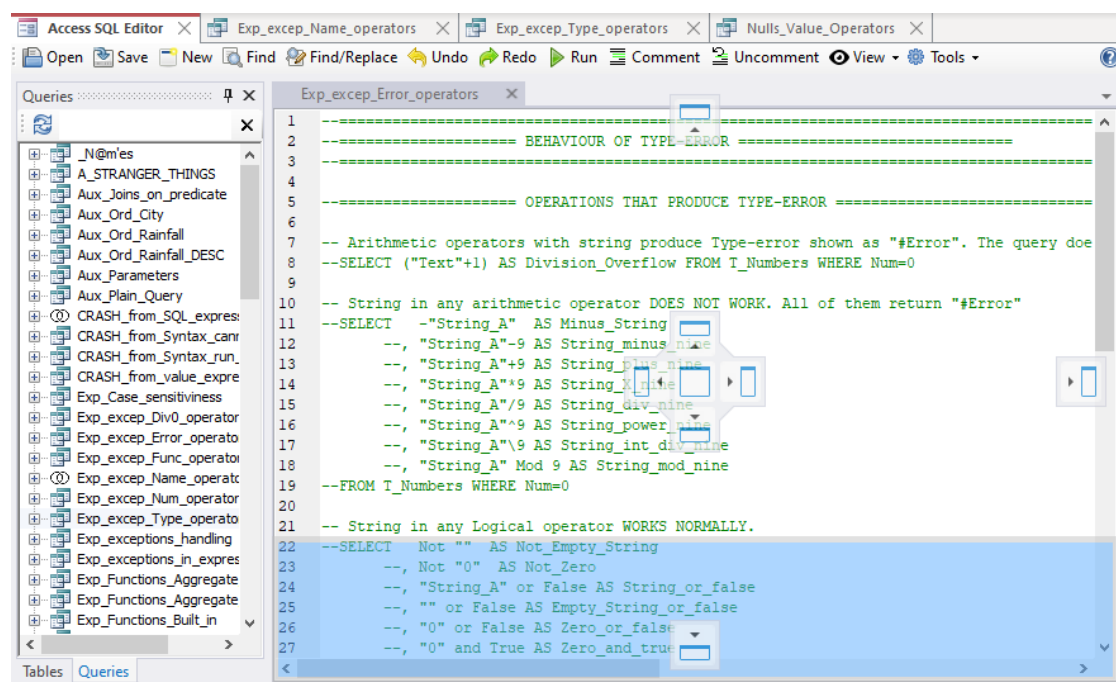
You can **resize** a **self-standing window** as you do with any other Microsoft Windows.

Finally, you can **move** a **self-standing window** by doing drag-and-drop on its thick top frame. You can also **move** any **sub-window** from the **self-standing window** by doing drag-and-drop on its thick top frame. Also, you can **move** any of the object panes from the **self-standing window** by doing drag-and-drop on the **pane's tab** at the **bottom** of its enclosing **sub-window**. If you want to know more about this, you may click "[F.5.6.9 How do I move an object pane, a sub-window or a self-standing window of the "Access SQL Editor"?](#)".

### F.5.6.9 How do I move an object pane, a sub-window or a self-standing window of the "Access SQL Editor"?

You **move** an **object pane** (click [F.5.2](#)), a **sub-window** (click [F.5.6.7](#)) or a **self-standing window** (click [F.5.6.8](#)) by doing drag-and-drop.

You **press** the left mouse button either on the **object pane tab**, on the **sub-window top frame**, or on the **self-standing window top frame**. Then, **without having released the button**, you **move** the mouse, and you will see a **blue shadow area**. This **blue shadow area** shows the **destination area** where the object will be **placed** if releasing the mouse button. The next screenshot shows the said **blue shadow area** (look at the bottom of the image):

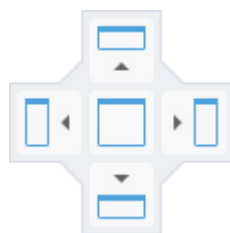


In addition to the **blue shadow area**, you will see **some individual positioning icons** inside the object area of the "**Access SQL Editor**". You may see some of the following **five individual positioning icons**: **left**, **right**, **center**, **top** and/or **bottom**. The **center individual positioning icon** is **only** shown when there is an **empty area** (gray background), within the object area. See the five icons below, in the same order as just listed:



In the screenshot above, you may see the **top**, **bottom**, and **right** individual positioning icons.

If you move the mouse (**without** releasing its button) **inside** a **pane-space** or **inside** a **sub-window**, a **combined positioning icon** will be shown (in addition to the individual ones already explained) in the center of the **pane-space** or **sub-window**. The next image shows the combined positioning icon.



In the screenshot above you can see the **combined positioning icon** in the center of the **pane-space**. The **combined positioning icon** contains **five** different squares: **left**, **right**, **center**, **top** and **bottom**.

If you move the mouse (**without** releasing its button) **inside** an **individual positioning icon**, or **inside** one of the **five squares** of a **combined positioning icon**, the **blue shadow area** will **change** to show the **destination area** corresponding to that specific **individual positioning icon** or **square** of a **combined positioning icon**.

**After moving** an **object pane** (click *F.5.2*), a **sub-window** (click *F.5.6.7*) or a **self-standing window** (click *F.5.6.8*) it **may change** to a different object type. If you want to know more about this, you may click *F.5.6.10*.

Finally, notice that you **cannot** directly move a **pane-space**. However, you can move **one** of its object panes, and **then** move **all the other ones** (manually, one by one) to the **same place** as the first one.

#### **F.5.6.10 How do I convert between an object pane, a pane-space, a sub-window, or a self-standing window in the “Access SQL Editor”?**

##### **How do I convert to a pane-space?**

You **convert** an **object pane**, a **sub-window** or a **self-standing window**, **into** a **pane-space** by **moving** it (click *F.5.6.9*) to either of the following places:

- The **center individual positioning icon** (click *F.5.6.9*) (which must correspond to an **empty area**).
  - For the case of an **object pane** or a **sub-window** it becomes a **new pane-space** in the destination location.
  - For the case of a **self-standing window**, **each** **sub-window** becomes a **new pane-space**, all of them placed in the destination location.
- The up, down, right or left squares of the **combined positioning icon** (click *F.5.6.9*) of an **existing pane-space**.
  - For the case of an **object pane** or a **sub-window**, it becomes a **new pane-space** in the destination location.
  - For the case of a **self-standing window**, **each** **sub-window** becomes a **new pane-space**, all of them placed in the destination location.
- The **center** square of the **combined positioning icon** (click *F.5.6.9*) of an **existing pane-space**.  
**All the object pane(s)** of the object being moved are **added** to the ones of the

existing pane-space.

#### How do I convert to a sub-window?

You **convert** an **object pane** or a **self-standing window**, into a **sub-window** by **moving** it (click *F.5.6.9*) to either of the following places:

- The up, down, right or left **individual positioning icon** (click *F.5.6.9*).
  - For the case of an **object pane** it becomes a **sub-window** in the destination location.
  - For the case of a **self-standing window**, each **sub-window** remains a **sub-window**, all of them placed in the destination location.
- The up, down, right or left **squares** of the **combined positioning icon** (click *F.5.6.9*) of an **existing sub-window**.
  - For the case of an **object pane** it becomes a **sub-window** in the destination location.
  - For the case of a **self-standing window**, each **sub-window** remains a **sub-window**, all of them placed in the destination location.
- The **center** square of the **combined positioning icon** (click *F.5.6.9*) of an **existing sub-window**.  
All the **object pane(s)** of the object being moved are **added** to the ones of the **existing pane-space**.

#### How do I convert to a self-standing window?

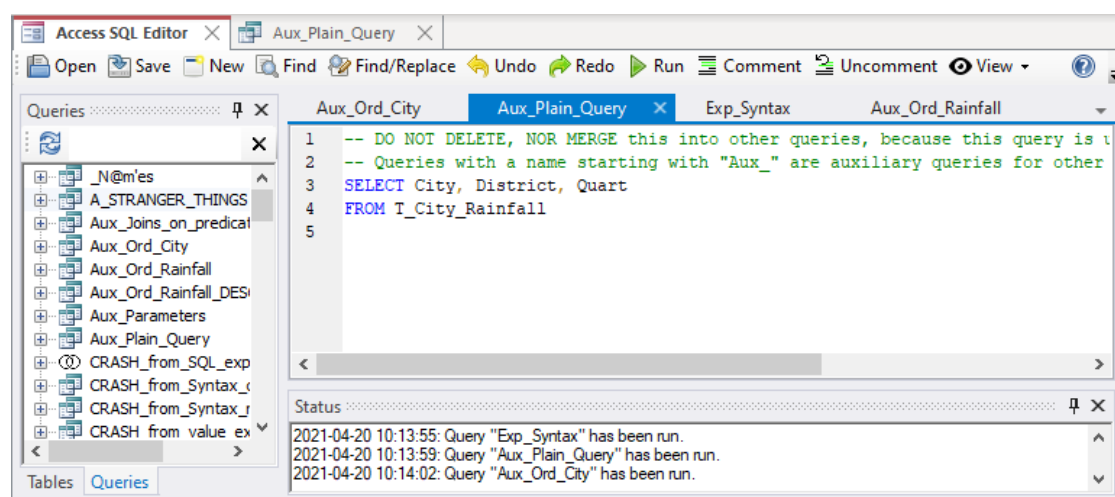
You **convert** an **object pane** or a **sub-window**, into a **self-standing window** by **moving** it (click *F.5.6.9*) to any place, **inside** or **outside** of the object area, that **is not** a **positioning icon** (click *F.5.6.9*).

#### F.5.6.11 What can be a good layout for the “Access SQL Editor”?

My preferred layout for the “**Access SQL Editor**” is to have:

- The “**Queries**” pane and the “**Tables**” pane together in a vertical sub-window, on the left side of the object area.
- All the query panes in a pane-space, at the top-right corner of the object area.
- The “**Status**” pane in a horizontal sub-window, at the bottom of the object area.
- The “**Status**” pane in a horizontal sub-window, at the bottom of the object area.
- A hidden “**Navigation Pane**” in the MS-Access window.

The next screenshot shows this “**Access SQL Editor**” layout that I prefer:



In addition to the above layout, using self-standing windows is very useful to view two (or more) Queries side by side. This allows you to compare their SQL code and/or to copy/paste code between them. When you are developing new Queries that are based on some existing Query(s), viewing them side by side, each in its own **self-standing window**, will be very valuable.

## F.6 What are the SQL operators I use to write my Queries?

SQL operators are the basic building blocks to write the SQL operations of a Query. There are **two groups** of SQL operators:

- **Consulting Operators:**  
They allow to **analyze** and **process** database Table data, presenting the results that you want. Table data remains unmodified.  
You may click “*F.6.1 What are the SQL consulting operations?*”.
- **Data-changing Operators:**  
They allow to **change** your database Table data in the way you want. They do not allow to analyze and process Table data.  
You may click “*F.6.2 What are the SQL data-changing operators?*”.

### F.6.1 What are the SQL consulting operations?

You can think of an SQL **consulting** operation as taking **one** or **two input record-lists** (plus other optional operands that are not a record-lists) and producing as a result **one output record-list**. There are only **four SQL consulting** operations:

- **Select operation:**  
**One operator** that has **several** (some optional) **clauses**.  
Its **clauses** are: “**SELECT**”, “**DISTINCT**”, “**DISTINCTROW**”, “**TOP**”, “**FROM**”, “**WHERE**”, “**GROUP BY**”, “**HAVING**” and “**ORDER BY**”.
- **Join operation:**  
**Four operators**, each having either **one** or **two clauses**.  
The **Join operators** are: “**,**”, “**INNER JOIN**”, “**RIGHT JOIN**” and “**LEFT JOIN**”. The first operator does not have the “**ON**” clause while the last **three**

Join operators have the “ON” clause.

- **Union operation:**  
Two operators, each having only one clause.  
The Union operators are: “UNION” and “UNION ALL”
- **Transform operation:**  
One operator that has several (some optional) clauses.  
Its clauses are: “TRANSFORM”, “SELECT”, “DISTINCT”, “DISTINCTROW”, “FROM”, “WHERE”, “GROUP BY”, “ORDER BY”, “PIVOT” and “IN”.

With only four consulting operations (and eight operators), you may think that SQL is much simpler than a conventional imperative programming language (e.g., VBA, C, Java, ...). However, SQL is not simple, because some SQL consulting operators have many options that make each of them much more complex than any statement from an imperative programming language. I now briefly explain the four SQL consulting operators:

- **Select operator:** “SELECT” clause plus other clauses  
You can think of this operator as taking one input record-list (plus other optional operands that are not record-lists) to produce one output record-list. **Optional step 1** consists of retaining certain input records. **Step 2** consists of modifying the (retained) input record-list to produce the output record-list. There are three modification options: a) produce one output record (changing fields and field values) from each (retained) input record; b) produce one output record from each group of (retained) input records; c) produce only one modified record from all (retained) input records. After record modification, there may be up to four additional optional steps. **Optional step 3** consists of retaining some of the aggregated output records. **Optional step 4** consists of discarding duplicates to produce distinct output records. **Optional step 5** consists of ordering the output record-list. **Optional step 6** consists of retaining only the top output records. The output record-list of a Select is therefore the (top) (ordered) (distinct) (retained) modified (retained) input record-list, where the words between parentheses represent optional processing steps.  
If you want to know more, you may click “F.7 [What is a Select operation and how do I write it?](#)”.

- **Join operators:** “,”, “INNER JOIN”, “RIGHT JOIN” and “LEFT JOIN”  
You may think of each of the four Join operators as taking two input record-lists (plus one more possible operand that is not a record-list) to produce one output record-list. Each output record has all the field values of one record from its left operand followed by all the field values of one record from its right operand. On some cases, the values from one of both records may become all Nulls. The “,” is called the Cross-Join operator. The “RIGHT JOIN” and “LEFT JOIN” are jointly called Outer-Join operators. There is also a Full-Outer-Join operation that does not have an MS-Access operator, but still is relevant to know.

If you want to know more, you may click “F.8 [What is a Join operation and how do I write it?](#)”.

- **Union operators:** “UNION” and “UNION ALL”  
You may think of each of the two Union operators as taking two input record-list to produce one output record-list. The output record-list is the mixing of all the



records from its two input record-lists. It is mandatory that the two input record-lists have the **same number of fields**. The “**UNION**” operator discards all redundant duplicate records in its **output** record-list, while the “**UNION ALL**” operator preserves all duplicates. Notice that “**UNION**” is **slower** than “**UNION ALL**” because the system has to check for duplicate records, and remove the redundant ones if found, before producing the **output** record-list.

If you want to know more, you may click “*F.9 What is a Union operation and how do I write it?*”.

- **Transform operator:** “**TRANSFORM**” clause plus other clauses  
This operator creates a **cross table** by converting the distinct **values** of an **expression** over its **input field** into **output field names** (i.e., **columns**) and presenting the **values** of **another expression** in the corresponding **cells** (**row x column**).

If you want to know more, you may click “*F.10 What is a Transform operation and how do I write it?*”.

## **F.6.2 What are the SQL data-changing operators?**

The three SQL **data-changing** operators are:

- **“DELETE” operator:**  
This operator allows you to **delete** the **records** you want from a given database Table.  
If you want to know more, you may click “*F.13.1 What is a Delete operation and how do I write it?*”.
- **“INSERT” operator:**  
This operator allows you to insert the records you want into a given database Table.  
If you want to know more, you may click “*F.13.2 What is an Insert operation and how do I write it?*”.
- **“UPDATE” operator:**  
This operator allows you to modify field values from the records that you want from a given database Table. This operator is somehow similar to a find-and-replace function from many applications (Word, Excel, ...).  
If you want to know more, you may click “*F.13.3 What is an Update operation and how do I write it?*”.

## **F.7 What is a Select operation and how do I write it?**

A **Select operation** is an SQL operation performed with a **Select** operator (click *F.7.1*) plus its corresponding operands. Therefore, a **Select operation** is the complete SQL code associated to the **Select operator**.

A simple example of a **Select operation**<sup>27</sup> is:

```
SELECT Capital AS Cap_City, (Temp_Min + Temp_Max)/2 AS Temp_Average
FROM T_Capital_Temps
WHERE Capital <> "Beijing"
```

<sup>27</sup> This is the Query “**F\_Select\_w\_no\_aggreg**” from file “**Company\_Database.accdb**”.

The above SQL operation **retains** the **input** records from the Table “**T\_Capital\_Temps**” (click *F.10.5*) in which the value of the field “**Capital**” is different from the constant “**Beijing**”. For each such **retained input** record it produces **one modified** record with a first **output** field named “**Cap\_City**” having the value of the first “**SELECT**” **expression** “**Capital**”, and a second **output** field “**Temp\_Average**” having the value of the second “**SELECT**” **expression** “**(Temp\_Min+Temp\_Max)/2**”. These **modified** records are no further processed, and they become the **output** record-list.

If you just want to know how to **write** and **run** a **simple Select operation**, you may click:

- “*A.4 How do I write and run my first SQL Query?*”
- “*A.6 How do I write and run my first Select Query with record aggregation?*”

If you want to **know more** about a **Select operation**, you may click:

- “*F.7.1 What is the Select operator?*”
- “*F.7.2 What are the three types of Select?*”
- “*F.7.3 What is the dataflow of a Select?*”
- “*F.7.4 What is the input record-list (“FROM” clause) of a Select?*”
- “*F.7.5 What are the output fields (“SELECT” clause) of a Select?*”
- “*F.7.6 What is the output record-list of a Select?*”
- “*F.7.7 What is the “WHERE” clause of a Select?*”
- “*F.7.8 What is the “DISTINCTROW” clause of a Select?*”
- “*F.7.9 What is the “GROUP BY” clause of a Select-group by aggreg?*”
- “*F.7.10 What is the “HAVING” clause of Select-group by aggreg or Select-total aggreg?*”
- “*F.7.11 What is the “DISTINCT” clause of a Select?*”
- “*F.7.12 How do I use “ORDER BY” to order the output records of a Select?*”
- “*F.7.13 What is the “TOP” clause of a Select?*”
- “*F.7.14 How do I write a correct (syntax) Select?*”
- “*F.7.15 How do I write a correct (syntax) Select-no aggreg?*”
- “*F.7.16 How do I write a correct (syntax) Select-group by aggreg?*”
- “*F.7.17 How do I write a correct (syntax) Select-total aggreg?*”
- “*F.7.18 What is an SQL aggregate function?*”

### **F.7.1 What is the Select operator?**

The most basic and fundamental SQL operator is **Select**. It is also called “**Select from where**”, to highlight that it almost always includes the clauses “**SELECT**”, “**FROM**” and “**WHERE**”.



**Select** is a **consulting** (click *F.6*) operator that works over **one** single **input** record-list (plus other optional operands that are not record-lists). It is **the most complex SQL operator** because it has **many** optional clauses and features.

The **Select** operator includes the clauses “**SELECT**”, “**DISTINCT**” (optional), “**TOP**” (optional), “**DISTINCTROW**” (optional and not advisable), “**FROM**”, “**WHERE**” (optional), “**GROUP BY**” (optional), “**HAVING**” (optional) and “**ORDER BY**” (optional).

A simplified view of writing (syntax) a **Select operation** is:

```
SELECT [DISTINCT] [TOP int [PERCENT] [DISTINCTROW]
      Output-expression() 1 to n
FROM Input-record-list
[ WHERE Where-Boolean-expression() ]
[ GROUP BY Group_by-expression() 1 to k ]
[ HAVING Having-Boolean-expression() ]
[ ORDER BY Order_by-expression() 1 to w ] ;
```

The clauses enclosed between square brackets “[ ]” are optional.

The **Select** operator works in the following steps:

1. **Retaining some input** records (optional “**WHERE**” clause)
2. **Modifying** the (retained) **input** record-list.  
(“**SELECT**” clause, plus, the optional “**GROUP BY**” clause and/or optionally SQL aggregate functions). This is the **output** record-list, **unless** one or more of the following **optional** steps is applied.
3. **Retaining some output** records (optional “**HAVING**” clause)
4. **Suppressing duplicate output** records (optional “**DISTINCT**” clause).
5. **Ordering** the **output** record-list (optional “**ORDER BY**” clause)
6. **Retaining the top output** records.

Therefore, the **output** record-list from the **Select** operator is the (top) (ordered) (distinct) (retained) **modified** (retained) **input** record-list, where words between parenthesis denote **optional** processing steps.

I now explain these six steps in more detail:

1. **Retaining some input** records (optional “**WHERE**” clause)  
Consists of **retaining** only **some input** records, while **discarding** the others.  
The result is the **retained input** record-list.  
The clause “**FROM**” (click *F.7.4*) indicates the **input** record-list.  
The optional “**WHERE**” clause (click *F.7.7*) has a *Boolean* expression that indicates what **input** records you want to **retain**. There is also an optional clause “**DISTINCTROW**” (click *F.7.8*), but my advice is you **do not** use it.
2. **Modifying** the (retained) **input** record-list  
Performed by the “**SELECT**” clause, plus, the optional “**GROUP BY**” clause and/or optional **SQL aggregate functions** (click *F.7.18*).  
Consists of **modifying** the (retained) **input** record to produce **different** records and possibly a different record-list.  
The result is the **output** record-list, still subject to the next four **optional** processing steps.  
The clause “**SELECT**” (click *F.7.5*) indicates the **fields, field values** and **field**

**names of output** records.

The optional clause “**GROUP BY**” (click *F.7.9*), combined with the optional usage of **SQL aggregate functions** (click *F.7.18*), indicates **one** out of **three** possible types (click *F.7.2*) of **record modification**: **no** aggregation, **group\_by** aggregation or **total** aggregation.

**3. Retaining some output** records (optional “**HAVING**” clause)

Consists of **retaining** only **some output** records, while **discarding** the others.

The *Boolean* expression in the optional “**HAVING**” (click *F.7.10*) clause indicates what **modified** records you want to **retain**. This clause can only be used when record aggregation took place in step 2.

**4. Suppressing duplicate output** records (optional “**DISTINCT**” clause)

Consists of classifying the **output** records in **disjoint sets** of records that have the **same values** in **all** their **output fields**. Out of **each set**, only **one output** record is retained, where this record **only** has the values of the **output fields**.

The optional clause “**DISTINCT**” (click *F.7.11*) indicates that the **Select** will produce **distinct output** records.

**5. Ordering the output** record-list (optional “**ORDER BY**” clause)

Consists of **ordering** the **output** record-list.

The optional clause “**ORDER BY**” (click *F.7.12*) indicates the record order that you want.

**6. Retaining the top output** records (optional “**TOP**” clause)

Consists of **retaining** a certain number or percentage of **top output** records. This step only makes sense if the optional **ordering** from **step 5** has been applied.

The optional clause “**TOP**” (click *F.7.12.1*) indicates **how many** (or what **percentage** of) top **output** records you want to **retain**.

Therefore, the **output** record-list from the **Select** operator is the (top) (ordered) (distinct) (retained) **modified** (retained) **input** record-list, where words between parenthesis denote optional features.

I think that the “**SELECT**” name is somehow misleading, because it seems as if you could only “**select**” some **input fields** and some **input records**. This is, as if you could **only** use the exact **input field names** as the **expressions** in the “**SELECT**” clause, and as if you could **not** do record aggregation. I think “**CREATE**” would have been a better name for the operator, because it signals that you can “**create**” new **output fields** and “**create**” new **output records**.

If you want to **see** these six steps in a **dataflow** diagram of a **Select** operation, showing also the different **clauses**, you may click “*F.7.3 What is the dataflow of a Select?*”.

If you just want to know **how to write and run** a **simple Select** operation, you may click “*A.4 How do I write and run my first SQL Query?*”.

If you just want to know **how to write and run** a **simple Select-group\_by\_agg**, you may click “*A.6 How do I write and run my first Select Query with record aggregation?*”.

If you want to know more about **Select**, you may click:

- “*F.7.2 What are the three types of Select?*”
- “*F.7.3 What is the dataflow of a Select?*”

- “F.7.4 What is the input record-list (“FROM” clause) of a Select?”
- “F.7.5 What are the output fields (“SELECT” clause) of a Select?”
- “F.7.6 What is the output record-list of a Select?”
- “F.7.14 How do I write a correct (syntax) Select?”

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “F.11.2 What are the SQL color codes used in this Guide?”.

## F.7.2 What are the three types of Select?

There are **three types** of **Select operations**, depending on the usage, or not, of the “GROUP BY” clause (click F.7.9) and/or of SQL aggregate functions (click F.7.18):

- “Select-no\_aggreg”  
It is a **Select** operation **without** a “GROUP BY” clause **and without SQL aggregate functions**.
- “Select-group\_by\_aggreg”  
It is a **Select** operation **with** a “GROUP BY” clause.
- “Select-total\_aggreg”  
It is a **Select** operation **without** a “GROUP BY” clause **but with SQL aggregate functions**.

Some relevant characteristics of the **three types** of **Select** operations are:

- “Select-no\_aggreg”
  - It **does not** have a “GROUP BY” clause, **nor SQL aggregate functions**:
  - It does **not** perform record **aggregation**.
  - It produces **one output record** for each (retained) **input record**.
  - The values of each **output** field are the result of an expression built over the “**Input-field-names**”.
- “Select-group\_by\_aggreg”
  - It **has** a “GROUP BY” clause.
  - It performs record aggregation based on the “GROUP BY” **expressions**.
  - It **may** have (and usually does have) **SQL aggregate functions**.
  - It classifies the (retained) **input** records in **disjoint record groups** and produces **only one output record** from each **record group**.
  - The values of **output** fields are computed by **expressions** built over the “GROUP BY” **expressions** and over **SQL aggregate functions** over expressions over the **fields** of the **input** record-list.
- “Select-total\_aggreg”
  - It **does not** have “GROUP BY” clause **but it has SQL aggregate functions**.
  - It performs **total** record **aggregation**.
  - It produces **only one**<sup>28</sup> **output record** for its whole **input** record-list. The values in the **output** fields are computed using **SQL aggregate functions** over expressions over the fields of the **input** record-list.

---

<sup>28</sup> If the **input** record-list is **empty**, then it produces **zero output** records.

If you want to know more about the **output** record-list of each of the **three types** of **Select** operation, you may click “[F.7.2 What are the three types of Select?](#)”.

The following table compares the **three types** of **Select operations** in terms of their having, or not, a “**GROUP BY**” clause and their having, or not, **SQL aggregate functions**:

		Has SQL aggregate functions?	
		No	Yes
Has a “GROUP BY” clause?	No	Select-no_aggreg	Select-total_aggreg
	Yes	Select-group_by_aggreg	

A simple example of a **Select-no\_aggreg**<sup>29</sup> operation is:

```
SELECT Capital AS Cap_City, (Temp_Min + Temp_Max)/2 AS Temp_Average
FROM T_Capital_Temps
WHERE Capital <> "Beijing"
```

This retains the records from the Table “**T\_Capital\_Temps**” (click [F.10.5](#)) in which the value of the field “**Capital**” is different from the constant “**Beijing**”, and **for each** such retained **input** record it produces **one output** record with the value of the “**Capital**” **input** field, and a field called “**Temp\_Average**” with the value of the average of the fields “**Temp\_Min**” and “**Temp\_Max**” from each corresponding **input** record.

A simple example of a **Select-group\_by\_aggreg**<sup>30</sup> operation is:

```
SELECT Capital AS Cap_City, Avg((Temp_Min + Temp_Max)/2) AS Temp_Average
FROM T_Capital_Temps
GROUP BY Capital
```

This creates **disjoint groups** of records, where **all** the records in the **same group** produce **the same value** of the “**GROUP BY**” **expression** “**Capital**”. It then aggregates all of the **input** records of **each disjoint group** into **one output** record. In each **output** record, it places the **average** of the expression “**(Temp\_Min + Temp\_Max) / 2**” computed over **all** the records in **each** of the disjoint **groups**.

A simple example of a **Select-total\_aggreg**<sup>31</sup> operation is:

```
SELECT Avg((Temp_Min + Temp_Max)/2) AS Temp_Average
FROM T_Capital_Temps
WHERE Capital <> "Beijing"
```

This **retains** the records from the Table “**T\_Capital\_Temps**” (click [F.10.5](#)) in which the value of the field “**Capital**” is different from the constant “**Beijing**”, and **for all such** retained **input** records it produces **one single output** record with a field called “**Temp\_Average**” with the value of the **average** of the expression “**(Temp\_Min + Temp\_Max) / 2**” from **all such input** records.

<sup>29</sup> This is the Query “**F\_Select\_w\_no\_aggreg**” from the “**Company\_Database.accdb**” file.

<sup>30</sup> This is the Query “**F\_Select\_w\_group\_by\_aggreg**” from the “**Company\_Database.accdb**” file.

<sup>31</sup> This is the Query “**F\_Select\_w\_total\_aggreg**” from the “**Company\_Database.accdb**” file.

If you want to know more about **SQL aggregate functions**, you may click “[F.7.18 \*What is an SQL aggregate function?\*](#)”.

If you just want to know **how to write and run** a simple **Select** operation, you may click “[A.4 \*How do I write and run my first SQL Query?\*](#)”.

If you just want to know **how to write and run** a simple **Select-group\_by\_aggreg**, you may click “[A.6 \*How do I write and run my first Select Query with record aggregation?\*](#)”.

If you want to know the **SQL color codes** used in this **Lightning Guide**, you may click “[F.11.2 \*What are the SQL color codes used in this Guide?\*](#)”.

### **F.7.3 What is the dataflow of a Select?**

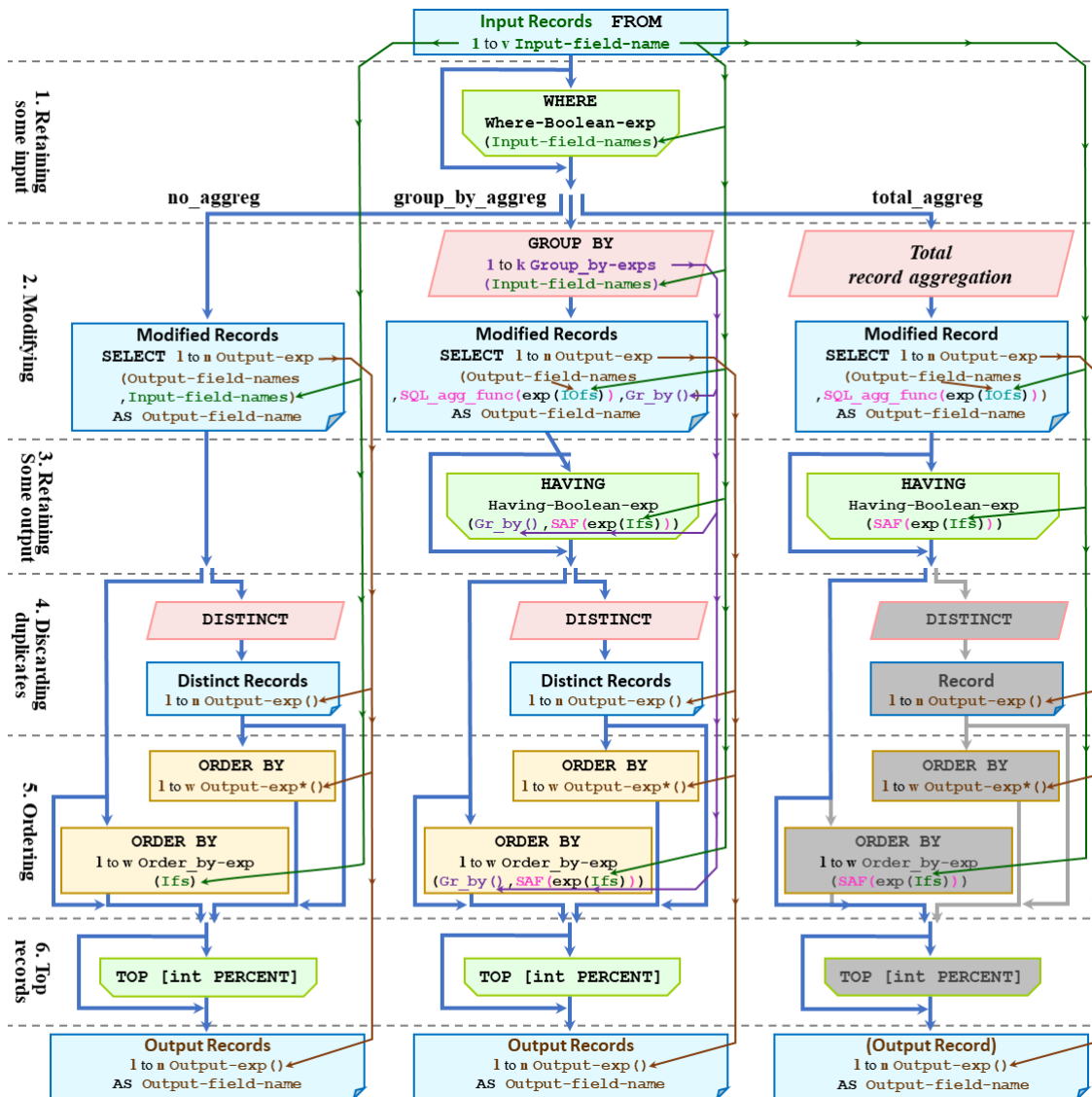
The picture below represents the **dataflow** for the three **types** of **Select** operations (click [F.7.3](#)). The **Select-no\_aggreg** is on the left, the **Select-group\_by\_aggreg** is in the middle, and the **Select-total\_aggreg** is on the right.

As was shown in [F.7.1](#), the **Select** dataflow has the following **six** processing **steps**:

1. **Retaining some input** records (optional “**WHERE**” clause)
2. **Modifying** the (retained) **input** record-list.  
(“**SELECT**” clause, plus, the optional “**GROUP BY**” clause and/or optionally SQL aggregate functions). This is the **output** record-list, **unless** one or more of the following **optional** steps is applied.
3. **Retaining some output** records (optional “**HAVING**” clause)
4. **Suppressing duplicate output** records (optional “**DISTINCT**” clause).
5. **Ordering** the **output** record-list (optional “**ORDER BY**” clause)
6. **Retaining the top output** records.

You may see that **each step** is delimited in the next picture with a **horizontal dashed line**, and a **label** on the leftmost side. You may also see that the **first step** is the **same** regardless of the **type** of **Select** operation, while the other **steps** have some differences between the **three types** of **Select** operation.

The picture representing the **dataflow** of the **Select** operator is:



The **blue rectangular boxes** with a folded corner (e.g., the ones labelled “**Modified Records**”) represent **record-lists**. I indicate inside **each** box some of the characteristics of the fields of the record-list, such as the expressions that determine the field values of each record.

The **green dented boxes** (e.g., “**WHERE**”) represent “**retaining**” clauses that **retain** some records, while **discarding** the others. These boxes **do not modify** the records themselves: each **retained** record remains exactly the same as it was before being processed by the green dented box. When applicable, I indicate inside each box the **Boolean** expression associated to the “**retaining**” clause.

The **pink trapezoidal boxes** (e.g., “**DISTINCT**”) represent “**aggregating**” clauses that **classify input** records in **disjoint groups**, producing **one aggregated record from each group**. These boxes **modify** the record-list and **may also** modify the record **values** (e.g., when aggregating **values** from each “**GROUP BY**” **group** of records).

- The box labelled “**GROUP BY**” performs **record aggregation** based on its associated “**GROUP BY**” expressions “**Group\_by-exp ()**” **1** to **k**.



- The box labelled “*Total record aggregation*” performs **total record aggregation**, this is, **only one modified** record is produced from **all** (retained) **input** records.
- The box labelled “**DISTINCT**” produces **distinct modified** records, which consists of **grouping** them when the values of all their **output** fields are the same (i.e., when the results of all their “**SELECT**” expressions “**Output-exp ()**” **1** to **n** are the same), and producing **only one** record out of **each group** of duplicate modified records.

The **yellow rectangular boxes** labelled “**ORDER BY**” represent the two variants (depending on the usage of “**DISTINCT**”) of the “**ORDER BY**” **ordering clause**. These boxes **do not modify** the records themselves: the records remain **exactly the same** as they were before being processed by the **yellow rectangular box**. When applicable, I indicate inside each box the **expressions** used to **order** the records.

When the **background** of any of the previous boxes is **shaded in gray** it means that the box is irrelevant. This happens in the dataflow for the “**total\_aggreg**” **Select** operation, placed on the right. The reason is that this type of **Select** operation aggregates **all input** records into **only one record** (which can be later reduced to **zero records** by the “**HAVING**” clause). This makes it irrelevant to apply, or not, the “**TOP**”, “**DISTINCT**” and/or “**ORDER BY**” clauses.

The **thick light blue** arrowed lines represent the **dataflow** across the **six** steps of the **Select** operator. The **dataflow** starts from the **blue rectangular box** with a folded corner labeled “**Input Records**” at the top, to the similar boxes labeled “**Output Record(s)**” at the bottom. Notice how these thick **light blue** arrowed lines depict **dataflow detours** to go **around** each of the several **optional steps** (optional clauses).

The **green dented box** at the top, labelled “**WHERE**”, represents the (optional) dataflow through the “**WHERE**” clause, that is performed equally in the **three types** of **Select** operation. From that point on, **each** of the **three** vertical **dataflows** is **specific** to **each** of the **three types** of **Select** operations. The **dataflow** for the “**no\_aggreg**” expression is on the left, the one for the “**group\_by\_aggreg**” is in the middle, and the one for the “**total\_aggreg**” is on the right.

The green text “**1 to v Input-field-name**” represents the “**v**” “**Input-field-names**” in the **input** record-list to the **Select** operation. The **green** text “**Ifs**” is a shorthand for “**Input-field-names**”.

The brown text “**Output-field-names**” represents the “**n**” **output field names** of the “**n**” fields in the **output** record-list.

The turquoise text “**IOfs**” is a shorthand for “**Input/Output-field-names**”. These are the “**v**” **Input-field-names**” and the “**n**” “**Output-field-names**”.

The purple text “**1 to k Group\_by-exp ()**” represents “**k**” “**GROUP BY**” **expressions** written after the “**GROUP BY**” keyword. Each of these **expressions** is built over the “**Input-field-names**”. The purple text “**Gr\_by ()**” is a shorthand to represent the “**GROUP BY**” **expressions**.

The brown text “**1 to n Output-exp ()**” represents the “**n**” “**SELECT**” **expressions** written after the “**SELECT**” keyword. These “**SELECT**” **expressions** are used to compute the “**n**” **field values** of **each output record**.

The black text “1 to w **Order\_by-exp ()**” represents the 1 to w the “**ORDER BY**” expressions written after the “**ORDER BY**” keyword. These “**ORDER BY**” expressions are used to **order** the (distinct) (retained) **modified** records.

The **terms** between the parentheses of both the “n” “**SELECT**” expressions “**Output-exp ()**”, and the “w” “**GROUP BY**” expressions “**Order\_by-exp ()**”, are the **elements** used to build them, by **combining** these **elements** with functions (excluding SQL aggregate), value operators and constants.

Notice how the **elements** used to build the “**Output-exp ()**” and the “**Order\_by-exp ()**” expressions are **different** depending on the **type** of **Select** operation. You may see the corresponding **elements** for the “**Output-exp ()**” by looking into the **three blue rectangular boxes** aligned horizontally near the **top** of the picture: the **three** boxes labelled “**Modified Record(s)**”.

You may see the corresponding **elements** for the “**Order\_by-exp ()**” by looking into the **three green dented boxes** aligned horizontally at the **bottom** of the picture: the **three** boxes labelled “**ORDER BY**” and having “**Order\_by-exp ()**” inside them.

Notice further how there are **two** sets of **three** “**ORDER BY**” **green dented boxes**. The ones **without** a **previous** “**DISTINCT**” box have “**Order\_by-exp ()**” expressions inside them, and I have just described them. The ones **with** a **previous** “**DISTINCT**” box have “**Output-exp\* ()**” expressions inside them. This means that when “**DISTINCT**” is used, the “**ORDER BY**” clause can **only** contain **exactly the same** “**SELECT**” expressions “**Output-exp ()**” that **do not include any** “**Output-field-name**”. This is why I added the “\*” to indicate that there is this additional restriction. Therefore, the term “**Output-exp\* ()**” denotes an “**Output-exp ()**” that **does not include any** “**Output-field-name**”.

The fuchsia text “**SQL\_agg\_func ()**” represents any of the SQL aggregate functions (click *F.7.18*). The fuchsia text “**SAF ()**” is a shorthand for “**SQL\_agg\_func ()**”.

The black text “**exp ()**” represents any given value **expression**.

The thin **green** arrowed lines allow to **track** where the “v” “**Input-field-names**” can be used as expression **elements**.

The thin **purple** arrowed lines allow to **track** where the “k” “**GROUP BY**” expressions “**Group\_by-exp ()**” can be used as expression **elements**.

The thin **brown** arrowed lines allow to **track where** the “n” “**SELECT**” expressions “**Output-exp ()**” can be used as **expression** elements.

I think you will find very illustrative to match this dataflow picture with the explanation that I provide in “*F.7.6 What is the output record-list of a Select?*”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “*F.11.2 What are the SQL color codes used in this Guide?*”.

#### **F.7.4 What is the input record-list (“FROM” clause) of a Select?**

In a **Select** operation, the mandatory “**FROM**” clause indicates the **input record-list**, as



follows:

```
FROM { [[ (] {Table-name or Query-name} [)] ]
      or [ {Table-name or Query-name} [AS Input-record-list-name] ]
      or [ ( {Select-opr or Union-opr} ) [AS Input-record-list-name] ]
      or [[ (] Inner-or-Outer-Join-opr [)] or Cross-Join-opr [ ] }
```

The **input** record-list of the **Select** operation is **written exactly the same** as in a **Transform** operation, but in a **different** way from the **Join** and **Union** operators.

The **input** record-list is written after the “**FROM**” keyword. It can be a **Table** name, a **Query** name, a **Join** operation, a **Select** operation or a **Union** operation.

The rules for **parentheses** and the “**AS**” clause depend on what is the **input** record-list after the “**FROM**” clause, as follows:

- **Table name or Query name**  
It **may** be enclosed between parentheses, or, it **may** have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one of the **Join** operation.
- **Select operation or Union operation**  
It **must** be enclosed between parentheses. It **may also** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the ones of the **Join** and the **Union** operations.
- **Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.
- **Cross-Join operation**  
It **must not** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.

If you want to see the **dataflow** diagram of the **three types** of **Select** operations, you may click “[F.7.3 What is the dataflow of a Select?](#)”.

### F.7.5 What are the output fields (“**SELECT**” clause) of a **Select**?

In a **Select** operation (click [F.7.1](#)), the mandatory “**SELECT**” clause determines the **output fields**, the **output field names**, the **output field order**, the **output data/field types** and the **output field values**, as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      { * or
        Output-exp_1(exp-elements) [AS Output-field-name_1]
      [, ...
        , Output-exp_n(exp-elements) [AS Output-field-name_n] ] }
```

Each “**SELECT**” expression “**Output-exp\_i()**” produces **one output field**. Therefore, the **number of output fields** is the **number of “SELECT” expressions**. In the code above, there are “**n**” **output fields**.

Notice that the **number “n” of output fields** is **totally independent**<sup>32</sup> of the **number of**

<sup>32</sup> Unless you are using “**SELECT \***”, which I advise you **do not** to use.

**input** fields (this is, the number of “**Input-field-names**”). For example, if the **input** record-list has **3** fields, there can be **1, 3, 11** or **any other number** of **output** fields. The number “**n**” of **output** fields is only determined<sup>33</sup> by the number of “**SELECT**” expressions “**Output-exp ()**”.

### What are the output field names of a Select?

If **after** any given “**SELECT**” **expression**, you write the keyword “**AS**” followed by a **name**, this will be the name of this **output** field.

Otherwise, if you are **not** using the “**AS**” clause, you then have two cases:

- If the “**SELECT**” **expression** consists **exactly** of a single **input field name**, then the **output field name** will be that **input field name**.
- If the “**SELECT**” **expression** is **anything other** than **exactly** a single **input field name**, MS-Access will assign to it the **output field name** “**ExprXXXX**”, where “**XXXX**” is a four-digit integer number.

Notice that for the first bullet point, if there are **more than one** “**SELECT**” **expression** that consist **exactly** of **the same** single **input field name**, then the **output field name** for **one of these output fields** will be that **input field name**, and for each of the other ones MS-Access will assign the **output field name** “**ExprXXXX**”, where “**XXXX**” is a four-digit integer number.

Notice also that if you use an **input field name** as an **output field name**, you must **qualify all** the usages of **that input field name** to avoid ambiguity. This is, if you write an **input field name** after the “**AS**” keyword, you must **qualify all** the usages of **that input field name** to avoid ambiguity. The ambiguity would arise from using the **same name** to denote **both an input and an output** field that are **different** (remind we are using the “**AS**” clause). Recall that qualifying a field name (click C.2.2) consists of prefixing it by the name of the record-list where it belongs with an intermediate period “.” character.

### What is the output field order of a Select?

The **output field order** corresponds to the order of the “**SELECT**” **expressions** “**Output-exp\_i ()**” **1 to n**.

### What are the output data/field types of a Select?

The **data/field type** of each **output field** “**i**” is the one of is corresponding “**SELECT**” **expression** “**Output-exp\_i ()**”, for “**i**” from **1** to **n**.

### What are the output fields, field names, field order and data/field types of a “SELECT \*”?

As an exception to the rules above, for the specific case of a **Select-no\_aggreg** (i.e., a **Select** without “**GROUP BY**” clause, and without SQL aggregate functions) it is possible to replace the list of “**SELECT**” **expressions** by just **one** asterisk character: “**SELECT \***”. I advise you do not use this option (click K.4.9).

If in spite of my advice you use “**SELECT \***”, then the **output fields**, the **output field names**, the **output field order** and the **output data types** will be **exactly the same as**

<sup>33</sup> Unless you are using “**SELECT \***”, which I advise you **do not** to use.

the **input** ones. If the **input** record-list is a **Query name**, the **output field order** will be the field order of the **input** Query in “**SQL View**” (and **not** the one in “**Datasheet View**” nor the one in “**Design View**”). If the **input** record-list is a **Table name**, the **output field order** and the **output field names** will be ones of the **input** Table in “**Design View**” (and **not** the ones in “**Datasheet View**”).

### What are the output field values and/or output record-list of a Select?

If you want to know what are the **output field values** and/or what is the **output record-list** of a **Select**, you may click “[F.7.6 What is the output record-list of a Select?](#)”.

### **F.7.6 What is the output record-list of a Select?**

You may click:

- “[F.7.6.1 What are the output field values of a Select-no aggreg?](#)”
- “[F.7.6.2 What are the output field values of a Select-group by aggreg?](#)”
- “[F.7.6.3 What are the output field values of a Select-total aggreg?](#)”
- “[F.7.6.4 What are the output records of a Select?](#)”
- “[F.7.6.5 How many output records does a Select produce?](#)”

If you rather want to know what are the **output fields** of a **Select**, you may click “[F.7.5 What are the output fields \(“SELECT” clause\) of a Select?](#)”.

#### **F.7.6.1 What are the output field values of a Select-no aggreg?**

A **Select-no aggreg** is a **Select** without the “**GROUP BY**” (click [F.7.9](#)) clause and without **SQL aggregate functions** (click [F.7.18](#)).

In a **Select-no aggreg**, the **output field values** are determined by the “**SELECT**” clause as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
{
  * or
  Output-exp_1(INOUT-field-names) [AS Output-field-name_1]
  [, ...
  , Output-exp_n(INOUT-field-names) [AS Output-field-name_n] ]}
```

In a **Select-no aggreg** **each** (retained) **input record** “**In<sub>j</sub>**” (that has “**v**” **input** fields), produces **one modified** record “**Mod<sub>j</sub>**”, with “**n**” **output** fields, where “**n**” is the number of “**SELECT**” **expressions**.

For each and every **modified** record “**Mod<sub>j</sub>**” the **value** of its **output** field “**i**” is the result of the “**SELECT**” **expression** “**Output-exp<sub>i</sub> ()**” computed over the “**v**” **input field values** of the **input record** “**In<sub>j</sub>**”, that produces the **modified record** “**Mod<sub>j</sub>**”.

In the “**SELECT**” clause above you can see that **each** “**Output-exp<sub>i</sub> ()**” is built over “**INOUT-field-names**”, this is, over the **1** to **v** “**Input-field-names**” and/or over the **1** to **n** “**Output-field-names**” (as long as you **do not** create a **circular** reference, click [F.7.14](#)). Since an “**Output-field-name-i**” is just a shorthand for its corresponding “**Output-exp<sub>i</sub> ()**”, once expanded, each “**Output-exp<sub>i</sub> ()**” is built **only** over the **1** to **v** “**Input-field-names**”.

For the **specific case** of the “**Select-no aggreg**”, you can write just “**\***” instead of the

list of “**Output-exps ()**”. If you do this, the list of **output field values** of modified record “**Mod\_j**” is **exactly the same** as the list of “**v**” **input field values** of its corresponding (retained) **input** record “**In\_j**”. My advice is you **do not** use an “\*”, because it is less readable than the explicit list of **input field names**, and it can also create some problems (click *K.4.9*).

The **output field values** in the **modified** records **will not be later changed** along the **Select** processing. However, some of the **modified** records may be **discarded** by the optional “**DISTINCT**” (click *F.7.11*) and/or “**TOP**” (click *F.7.13*) clauses. **Discarded modified** records will obviously **not** appear in the **output** record-list of the **Select**.

If you want to know about related concepts, you may click:

- “*F.7.5 What are the output fields (“SELECT” clause) of a Select?”*”
- “*F.7.6.2 What are the output field values of a Select-group by aggreg?”*”
- “*F.7.6.3 What are the output field values of a Select-total aggreg?”*”
- “*F.7.6.4 What are the output records of a Select?”*”
- “*F.7.6.5 How many output records does a Select produce?”*”

### **F.7.6.2 What are the output field values of a Select-group by aggreg?**

A **Select-group\_by\_aggreg** is a **Select** with the “**GROUP BY**” (click *F.7.9*) clause.

In a **Select-group\_by\_aggreg**, the **output field values** are determined by the “**SELECT**” and “**GROUP BY**” clauses as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , Group_by-exp_1(Input-field-names)
                  , ...
                  , Group_by-exp_k(Input-field-names)
                  , SQL_agg_func(exp_o11(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_o1y(INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
[ , ...
  , Output-exp_n( Output-field-names
                , Group_by-exp_1(Input-field-names)
                , ...
                , Group_by-exp_k(Input-field-names)
                , SQL_agg_func(exp_on1(INOUT-field-names))
                , ...
                , SQL_agg_func(exp_onz(INOUT-field-names))
                ) [ AS Output-field-name_n ] ]
...
GROUP BY      Group_by-exp_1(Input-field-names)
[ , ...
  , Group_by-exp_k(Input-field-names) ]
```

In a **Select-group\_by\_aggreg**, each of the (retained) **input** records is **assigned** to one disjoint **group** “**GIn\_j**”, and **one modified** (i.e., aggregated) record “**Mod\_j**” is produced out of **each** “**GIn\_j**” **group**. The (retained) **input** records in **each disjoint group** “**GIn\_j**” are the ones that produce **exactly the same results** in all the **1** to **k** “**GROUP BY**” expressions “**Group\_by-exp (Input-field-names)**”.

For each and every **modified** record “**Mod\_j**” the value of its “**i**” field is the result of “**SELECT**” **expression** “**Output-exp\_i ()**” computed over the “**v**” field values of **all**

the **input records** in the **group** “GIn\_j”, that produces the **modified** record “Mod\_j”. Each “**SELECT**” **expression** can be built over the following **elements**:

- **Other** “**Output-field-names**” (as long as you **do not** create a **circular** reference, click *F.7.14*).
- Any number of **exactly the same** “**GROUP BY**” **expressions** computed over the field values of **any** of the **input** records in **group** “GIn\_j”.

Notice that **all** the **input** records in **each group** **must** (as I indicated slightly above) produce **the same values** in **all** the **1** to **k** “**GROUP BY**” **expressions**. Therefore, we can compute the result of each “**GROUP BY**” **expression** over **any** of the **input** records in **group** “GIn\_j” and we would always get the same result.

- Any number of **SQL aggregate functions** “**SQL\_agg\_func()**”, each computed over its specific **expression** over the “**INOUT-field-names**”. This is, over the **1** to **v** “**Input-field-names**” and/or over the **1** to **n** “**Output-field-names**” (as long as you **do not** create a **circular** reference, **nor** a **nested** SQL aggregate function, click *F.7.14*). Since an “**Output-field-name\_x**” is just a shorthand for its corresponding “**SELECT**” **expression**, once expanded, each “**exp\_oij()**” above is built **only** over the **1** to **v** “**Input-field-names**”.

The **expression** that is the argument of each **SQL aggregate function** is computed over the “**v**” **field values** of **each and every** of the **input** records in the **group** “GIn\_j”. Each computation produces a given result, so the **overall** result is a **group of values**, **each** value coming from **each input** record in the **group** “GIn\_j”. The **SQL aggregate function** produces **one** value out of **one group** of values. For example, the “**Max()**” **SQL aggregate function** returns the maximum value in the **group** of values. Therefore, **each** **SQL aggregate function** computed over **expressions** over the “**Input-field-names**” of the records in **each group** “GIn\_j” will be computed over the corresponding **group** of values and will return **one** specific value for the **input record group** “GIn\_j”.

Therefore, **each** “**GROUP BY**” **expression** and each **SQL aggregate function** will return **one** coherent value from **each group** of **input records** “GIn\_j”. These are the values used to compute each “**SELECT**” **expression** “**Output-exp\_i()**”.

In short, for each and every **modified** record “Mod\_j”, produced from **input** record **group** “GIn\_j” the **value** of its field “i” is the result of “**SELECT**” **expression** “**Output-exp\_i()**” applied over **elements univocally computed** from the records in the **group** “GIn\_j”.

The **output field values** in the **modified** records **will not be later changed** along the **Select** processing. However, some of the **modified** records may be **discarded** by the optional “**HAVING**” (click *F.7.10*), “**DISTINCT**” (click *F.7.11*) and/or “**TOP**” (click *F.7.13*) clauses. **Discarded modified** records will obviously **not** appear in the **output** record-list of the **Select**.

Be aware that an expression “**exp\_oij()**” that is the argument of a given **SQL aggregate function** **cannot** contain any **SQL aggregate function**. This **cannot** happen in



a direct manner, **nor through a reference** to another “**Output-field-name**”.

If you want to know about related concepts, you may click:

- “F.7.5 What are the output fields (“SELECT” clause) of a Select?”
- “F.7.6.1 What are the output field values of a Select-no\_aggreg?”
- “F.7.6.3 What are the output field values of a Select-total\_aggreg?”
- “F.7.6.4 What are the output records of a Select?”
- “F.7.6.5 How many output records does a Select produce?”

### F.7.6.3 What are the output field values of a Select-total\_aggreg?

A **Select-total\_aggreg** is a **Select** without the “**GROUP BY**” (click F.7.9) clause but with SQL aggregate functions (click F.7.18).

In a **Select-total\_aggreg**, the **output field values** are determined by the “**SELECT**” clause as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , SQL_agg_func(exp_o11(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_o1y(INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
[ , ...
  , Output-exp_n( Output-field-names
                  , SQL_agg_func(exp_on1(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_onz(INOUT-field-names))
                  ) [ AS Output-field-name_n ] ]
```

In a **Select-total\_aggreg**, only **one** modified (i.e., aggregated) record “**Mod**” is produced out of **all** the (retained) **input** records.

For this **modified** record “**Mod**” the value of its “**i**” field is the result of “**SELECT**” expression “**Output-exp\_i()**” computed over the “**v**” field values of **all** the **input records**. Each “**SELECT**” expression can be built over the following elements:

- **Other “Output-field-names”** (as long as you **do not** create a **circular** reference, click F.7.14).
- Any number of SQL aggregate functions “**SQL\_agg\_func()**” each having as argument its specific **expression** over the “**INOUT-field-names**”. This is, over the **1** to **v** “**Input-field-names**” and/or over the **1** to **n** “**Output-field-names**” (as long as you **do not** create a **circular** reference, **nor** a **nested** SQL aggregate function, click F.7.14). Since an “**Output-field-name\_x**” is just a shorthand for its corresponding “**SELECT**” **expression**, once expanded, each “**exp\_oij()**” above is built **only** over the **1** to **v** “**Input-field-names**”.

The **expression** that is the argument of each SQL aggregate function is computed over the “**v**” **field values** of **each and every** of the **input** records. Each computation produces a given result, so the **overall** result is a **group of values**, **each** value coming from **each input** record. The SQL aggregate function produces **one** value out of **one group** of values. For example, the “**Max()**” SQL aggregate function returns the maximum value in the **group** of values.

Therefore, **each SQL aggregate function** computed over **expressions** over the “**Input-field-names**” of **all** the **input** records will be computed over the corresponding **group** of values and will return **one** specific value for **all** the **input** records.

Therefore, **each SQL aggregate function** will return **one** coherent value from all the (retained) **input** records. These are the values used to compute each “**SELECT**” **expression** “**Output-exp<sub>i</sub> ()**”.

In short, the **value** of each field “**i**” of the **only** modified “**Mod**” record is the result of “**SELECT**” **expression** “**Output-exp<sub>i</sub> ()**” computed over **elements univocally computed** from all the (retained) **input** records.

The **output field values** in the **modified** records **will not be later changed** along the **Select** processing. However, some of the **modified** records may be **discarded** by the optional “**HAVING**” (click *F.7.10*) clause. **Discarded modified** records will obviously **not** appear in the **output** record-list of the **Select**.

If you want to know about related concepts, you may click:

- “*F.7.5 What are the output fields (“SELECT” clause) of a Select?*”
- “*F.7.6.1 What are the output field values of a Select-no\_aggreg?*”
- “*F.7.6.2 What are the output field values of a Select-group\_by\_aggreg?*”
- “*F.7.6.4 What are the output records of a Select?*”
- “*F.7.6.5 How many output records does a Select produce?*”

#### **F.7.6.4 What are the output records of a Select?**

The **output** record order is **unknown**, unless you use the optional “**ORDER BY**” clause (click *F.7.12*).

The **output** records of a **Select** operation are the result of a **sequence** of processing **steps** (click *F.7.1*), many of them **optional**, starting from the **input** records, as follows.

##### **Input record-list**

The **input record-list** of the **Select** operation is written right after the “**FROM**” keyword (click *F.7.4*).

The “**Input-field-names**” are **all** the “**v**” **input** field names from the **input** record-list.

##### **Step 1. Retaining some input records (optional)**

If you use the optional clause “**WHERE**” (click *F.7.7*), it **retains** some **input** records, while **discarding** the others. The **retained input** records are the ones that produce **True** in the **Boolean** expression “**Where-Boolean-exp ()**” of the “**WHERE**” clause.

##### **Step 2. Modifying the (retained) input record-list**

The **modification** performed on the (retained) **input** record-list depends on the **type** of **Select** operation (click *F.7.2*), as follows:

- **Select-no\_aggreg**  
This is when you **do not** use the optional “**GROUP BY**” (click *F.7.9*) clause, **nor**

**SQL aggregate functions** (click *F.7.18*).

In this case, **one modified** record will be produced from **each** (retained) **input** record.

- **Select-group\_by\_aggreg**

This is when you **use** the optional “**GROUP BY**” (click *F.7.9*) clause.

In this case, **each** (retained) **input** record is **assigned** to **one** disjoint **group**, and **one modified** (i.e., aggregated) **record** is produced out of **each group**. The (retained) **input** records in **each disjoint group** are the ones that produce **exactly the same results** in **all** the “**GROUP BY**” **expressions** “**Group\_by-exp (Input-field-names)**” 1 to **k**.

- **Select-total\_aggreg**

This is when you **do not** use the optional “**GROUP BY**” clause (click *F.7.9*), but you **do use SQL aggregate functions** (click *F.7.18*).

In this case, **only one modified** (i.e., aggregated) **record** is produced out of **all** the (retained) **input** records.

### Step 3. Retaining some modified records (optional)

If you use the optional clause “**HAVING**” (click *F.7.10*), it **retains** some **modified** records, while **discarding** the others. The “**HAVING**” clause **cannot** be used in a **Select-no\_aggreg**.

The **retained modified** records are the ones that return **True** in the **Boolean** expression “**Having-Boolean-exp ()**”.

### Step 4. Generating the distinct (retained) modified record-list (optional)

If you use the optional clause “**DISTINCT**” (click *F.7.11*) only **one** modified record will be produced from **each set** of duplicate **modified** records. These are the **distinct** (retained) **modified** records.

### Step 5. Ordering the (distinct) (retained) modified record-list (optional)

If you use the optional clause “**ORDER BY**” (click *F.7.12*), the record-list from the previous step will be ordered according to the list of “**ORDER BY**” **expressions** “**Order\_by-exp ()**”. If you do not use the “**ORDER BY**” clause, the order of **output** records will be unknown.

### Step 6. Retaining top (ordered) (retained) modified records (optional)

If you use the optional “**TOP**” clause (click *F.7.13*) this will **retain** the top “**int+**” (or top “**int+**” percent if you use “**PERCENT**”) **output** records. This only makes sense if you are also using the “**ORDER BY**” clause, because otherwise the record order is unknown, and therefore, the top “**x**” fields are unpredictable.

### Output record-list

Therefore, the **output** record-list from the **Select** operator is the (top) (ordered) (distinct) (retained) **modified** (retained) **input** record-list, where words between parenthesis denote optional features.

If you want to **see** the **dataflow** diagram of the **three** types of **Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.



If you want to know about related concepts, you may click:

- “F.7.5 What are the output fields (“SELECT” clause) of a Select?”
- “F.7.6.1 What are the output field values of a Select-no aggreg?”
- “F.7.6.2 What are the output field values of a Select-group\_by aggreg?”
- “F.7.6.3 What are the output field values of a Select-total aggreg?”
- “F.7.6.5 How many output records does a Select produce?”

### **F.7.6.5 How many output records does a Select produce?**

Knowing how many records a **Select operation** produces is **very** useful when debugging your Queries, and also for **better** understanding how a **Select** works.

The **number** of **output** records of a **Select** operation is the result of a **sequence** of processing **steps** (click *F.7.1*), many of them **optional**, starting from the **number** of **input records**, as follows.

#### **Step 1:**

- If you use “**WHERE**” (click *F.7.7*), the **number** of records is the number of **input** records that **produce True** in the “**WHERE**” **Boolean** expression “**Where-Boolean-exp()**”.
- If you **do not** use “**WHERE**”, the **number** of records is the **number** of **input** records.

**Step 2:** The **number** of records depends on the **type** of **Select** (click *F.7.2*) as follows:

- If it is a **Select-no\_aggreg**, the **number** of records from the previous **step** is unchanged.
- If it is a **Select-group\_by\_aggreg**, the **number** of records is the **number** of **different arrays** of “**k**” **results** produced by the records from the previous **step** in **all** the **1** to **k** “**GROUP BY**” **expressions** “**Group\_by-exp(Input-field-names)**”.
- If it is a **Select-total\_aggreg**, the **number** of records is the lesser between **one** and the **number** of records from the previous **step**.

#### **Step 3:**

- If you use “**HAVING**” (click *F.7.10*), the **number** records is the **number** of records from the previous **step** that **produce True** in the “**HAVING**” **Boolean** expression “**Having-Boolean-exp()**”. Recall that “**HAVING**” **cannot** be used in a **Select-no\_aggreg**.
- If you **do not** use “**HAVING**”, the **number** of records from the previous **step** is unchanged.

#### **Step 5:**

- If you use “**DISTINCT**” (click *F.7.11*), the **number** records is the **number** of records from the previous **step** that **have different values in all fields**.
- If you **do not** use “**HAVING**”, the **number** of records from the previous **step** is unchanged.

**Step 6:**

- If you use “**TOP int**” (click *F.7.13*), the **number** of records is the **lesser between** the **number** of records from the previous **step and** “**int+**” records.
- If you use “**TOP int PERCENT**” (click *F.7.13*), the **number** of records is “**int+**” **percent of the number** of records from the previous **step**.
- If you use neither “**TOP**”, nor “**TOP PERCENT**” (click *F.7.13*), the **number** of records is the **number** of records from the previous **step**.

Notice that I have written “**int+**” above, instead of “**int**”, because in case **the record** in ordered position “**int**” (or “**int**” percent) has **the same values** in **all** the “**ORDER BY**” expressions as other “**q**” records, then **all** the “**q**” records will **also** be **output** records from the “**TOP**” clause.

If you want to know about related concepts, you may click:

- “*F.7.5 What are the output fields (“SELECT” clause) of a Select?*”
- “*F.7.6.1 What are the output field values of a Select-no aggreg?”*”
- “*F.7.6.2 What are the output field values of a Select-group by aggreg?”*”
- “*F.7.6.3 What are the output field values of a Select-total aggreg?”*”
- “*F.7.6.4 What are the output records of a Select?”*”

**F.7.7 What is the “WHERE” clause of a Select?**

In a **Select** operation (click *F.7.1*), the optional “**WHERE**” clause indicates what are the **retained input** records, as follows:

```
WHERE Where-Boolean-exp(Input-field-names)
```

The “**WHERE**” keyword is followed by a **Boolean expression**. This “**WHERE**” **Boolean expression** is built by **combining** the “**Input-field-names**” with functions (excluding SQL aggregate), value operators and constants.

The “**WHERE**” clause **retains** the **input** records whose field values produce **True** in the “**WHERE**” **Boolean** expression.

If a “**WHERE**” **Boolean** expression produces **Null**, the Query works normally. The **input** records producing **Null** are **not** retained.

If a “**WHERE**” **Boolean** expression produces an **exception** (e.g., divide by zero), on most cases the Query will crash.

Very important to highlight that if the **input** record-list is a Table name, no records will be **deleted** from the Table, and the Table remains **unmodified**. This is so because all the SQL **consulting** operators work over an **image** of Table’s records, and **not** over the Table records themselves. Remind that the **Select** operator is a **consulting** SQL operator (i.e., one that can only consult Tables) and it is not a **data-changing** SQL operator. If you want to actually **change** your Table’s records, you may click “*F.6.2 What are the SQL data-changing operators?*”.

If you want to **see** the **dataflow** diagram of the **three types** of **Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.

If you want to write (syntax) a correct “**WHERE**” clause, you may click *F.7.14*.

### **F.7.8** What is the “**DISTINCTROW**” clause of a **Select**?

My advice is you **do not use** the “**DISTINCTROW**” optional clause in **Select** operations because it is not compatible with SQL Server syntax.

You can get the same functionality by using the “**DISTINCT**” or “**UNION**” clauses (as required) in the **enclosed** SQL operation of the **Select** operation.

If you still want to use it, the “**DISTINCTROW**” clause discards **duplicate records** from the **input** record-list. Discarding **duplicate** records means leaving just **one input** record from each set of **duplicate input** records, while non-duplicate **input** records stay the same. Therefore, the **retained input** record-list will have **no duplicate** records.

Remind that two records are considered duplicate if they have **exactly the same values** when comparing all their fields values, field to field. Remind that for the purpose of discarding duplicate records **Nulls** are considered the same value.

You **cannot** use both the “**DISTINCT**” and “**DISTINCTROW**” clauses in the same **Select** operation.

The “**DISTINCTROW**” clause works the same for the **three types** of **Select** operations (click *F.7.2*).

Remind that the **retained input** record-list, or **retained input** records, are the list of **input** records **retained for further processing** after applying the “**WHERE**” and/or the “**DISTINCTROW**” clauses. The **input** records that are **not retained** for further processing are **discarded**.

Very important to highlight that if the **input** record-list is a Table name, no records will be **deleted** from the Table, and the Table remains **unmodified**. This is so because all the SQL **consulting** operators work over an **image** of Table’s records, and **not** over the Table records themselves. Remind that the **Select** operator is a **consulting** SQL operator (i.e., one that can only consult Tables) and it is not a **data-changing** SQL operator. If you want to actually **modify** your Table’s records, you may click “*F.6.2 What are the SQL data-changing operators?*”.

### **F.7.9** What is the “**GROUP BY**” clause of a **Select-group by aggreg**?

In a **Select-group\_by\_aggreg** (click *F.7.2*) operation, the mandatory “**GROUP BY**” clause indicates how to **classify** the (retained) **input** records into disjoint **groups**, as follows:

```
GROUP BY      Group_by-exp_1(Input-field-names)
             [ , ...
             , Group_by-exp_k(Input-field-names) ]
```

As you may see, the “**GROUP BY**” keyword is followed by a **list** of **expressions** “**Group\_by-exp(Input-field-names)**” **1** to **k**. Each such “**GROUP BY**” **expression** is built by **combining** the “**Input-field-names**” with functions (**excluding** SQL aggregate), value operators and constants.

Using “**GROUP BY**” causes that each (retained) **input** record is **assigned** to one disjoint **group**, and **one modified** (i.e., **aggregated**) **record** is produced out of **each group**. The (retained) **input** records **grouped** in **each disjoint group** are the (retained) **input**

records that produce **exactly the same results** in all the “GROUP BY” **expressions** “Group\_by-exps ()” 1 to k.

The “GROUP BY” **expressions** are extremely important, because they are, in a **Select-group\_by\_aggreg**, some of the **elements** that can be used to build the “SELECT” **expressions**, as well as the “HAVING” **Boolean expression** and the “ORDER BY” **expressions** (click *F.7.16*).

You **cannot** assign an identifier (with an “AS” clause) to individual “GROUP BY” **expressions**. Therefore, you have to write them **as such** when you use them as **elements** to build the “SELECT” **expressions** (click *F.7.5*), the “HAVING” **Boolean expression** (click *F.7.10*) and/or the “ORDER BY” **expressions** (click *F.7.16*). This may be cumbersome when a “GROUP BY” **expression** is large, but there is no way to avoid this.

If a “GROUP BY” **expression** produces **Null**, the Query works normally. Records producing **Null** in a “GROUP BY” **expression** will be **considered** as producing **the same** valid value in this **expression** and will be **assigned** to a **group** according to the normal grouping rules.

If a “GROUP BY” **expression** produces an **exception** (e.g., divide by zero), on most cases the Query will crash.

You can **only** use the optional “GROUP BY” clause in a **Select-group\_by\_aggreg** (click *F.7.2*). In a **Select-no\_aggreg** or in a **Select-total\_aggreg** you **cannot** use the “GROUP BY” clause.

If you want to see the **dataflow** diagram of the **three types** of **Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.

If you want to write (syntax) a correct “WHERE” clause, you may click *F.7.14*.

### **F.7.10 What is the “HAVING” clause of Select-group by aggreg or Select-total aggreg?**

In a **Select-group\_by\_aggreg** or **Select-total\_aggreg** (click *F.7.2*) operations, the optional “HAVING” clause indicates the **retained modified** records, as follows:

**HAVING** Having-Boolean-exp(having-elements)

As you may see, the “HAVING” keyword is followed by a **Boolean expression**. This “HAVING” **Boolean expression** is built by **combining** some **elements** (called “having-elements” above) with functions (**excluding** SQL aggregate), value operators and constants. The specific “having-elements” that can be used to build the “HAVING” **Boolean expression** depend on the type of **Select** operation, as I explain further below.

The “HAVING” clause **retains** the **modified** records that produce **True** in the “HAVING” **Boolean expression**.

If a “HAVING” **Boolean expression** produces **Null**, the Query works normally. The **modified** records producing **Null** are not retained.

If a “HAVING” **Boolean expression** produces an **exception** (e.g., divide by zero), on most cases the Query will crash.

You can use the “**HAVING**” and/or “**DISTINCT**” and/or “**TOP**” clauses together in a **Select** operation. Remind that if you use “**HAVING**” and/or “**DISTINCT**” plus “**TOP**”, the “**TOP**” clause will be applied **last** (click *F.7.1*).

You can **only** use the optional “**HAVING**” clause in either a **Select-group\_by\_aggreg** or in a **Select-total\_aggreg** (click *F.7.2*). In the **Select-no\_aggreg** you **cannot** use the “**HAVING**” clause.

If you want to see the **dataflow** diagram of the **three types** of **Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.

If you want to write (syntax) a correct “**WHERE**” clause, you may click *F.7.14*.

### What are the “having-elements” in a **Select-group\_by\_aggreg**?

In a **Select-group\_by\_aggreg**, the “**having-elements**” used to build the “**HAVING**” **Boolean expression** are the “**GROUP BY**” **expressions**, and any number of **SQL aggregate functions** each having as argument its specific **expression** over the “**Input-field-names**”.

You **cannot** assign an identifier (with an “**AS**” clause) to individual “**GROUP BY**” **expressions**. Therefore, you have to write them as **such** when you use them as **elements** to build the “**HAVING**” **Boolean expression**. This may be cumbersome when a “**GROUP BY**” **expression** is large, but there is no way to avoid this.

Summarizing, the way in which you can write the “**HAVING**” clause in a **Select-group\_by\_aggreg** is:

```
HAVING Having-Boolean-exp(  Group_by-exp_1(Input-field-names)
                          /  ...
                          /  Group_by-exp_k(Input-field-names)
                          /  SQL_agg_func(exp_h1(Input-field-names))
                          /  ...
                          /  SQL_agg_func(exp_ht(Input-field-names)) )
```

### What are the “having-elements” in a **Select-total\_aggreg**?

In a **Select-total\_aggreg**, the “**having-elements**” used to build the “**HAVING**” **Boolean expression** are any number of **SQL aggregate functions** each having as argument its specific **expression** over the “**Input-field-names**”.

Summarizing, the way you can write the “**HAVING**” clause in a **Select-total\_aggreg** is:

```
HAVING Having-Boolean-exp(  SQL_agg_func(exp_h1(Input-field-names))
                          /  ...
                          /  SQL_agg_func(exp_ht(Input-field-names)) )
```

## F.7.11 What is the “**DISTINCT**” clause of a **Select**?

In a **Select** operation (click *F.7.1*), the optional “**DISTINCT**” clause produces one **distinct** (i.e., with unique field values) record out of each **set** of (retained) **modified** records that have the same field values, as follows:

```
SELECT DISTINCT [TOP int [PERCENT]] [DISTINCTROW or ALL]
```

If you write the optional “**DISTINCT**” clause, the modified (retained) **input** records are **grouped** in disjoint sets, such that the records in each set have exactly the **same values** in **all** their **output fields**. From **each** such set, **only one** record is produced. This **only**



the **output** field **values**. Therefore, the record-list after the “**DISTINCT**” clause will have no duplicate **records**.

Remind that for the purpose of discarding duplicate records **Nulls** are considered the same value.

Notice that the records resulting from the “**DISTINCT**” clause **only** have the **output** field values. All other **information** existing in the modified records (e.g., the **values** of the **input** fields, or the values of the “**Group\_by-exp()**” **expressions**) **is lost**. This has an impact on the “**ORDER BY**” clause as follows:

- If you **use** the “**DISTINCT**” clause, each expression used in the “**ORDER BY**” clause **must be exactly the same** as one of the “**SELECT**” **expressions** that do not contain any “**Output-field-name**”. In other words, each “**Order\_by-exp-i()**” **must be exactly the same** as one of the “**Output-exp-x()**” that do not contain any “**Output-field-name**”. If you want to overcome this restriction, you can just remove the “**ORDER BY**” clause from this **Select** operation and enclose it in **another Select** operation (without the “**DISTINCT**” clause), in which you can write the “**ORDER BY**” expressions that you want.
- If you **do not** use the “**DISTINCT**” clause, the “**ORDER BY**” expressions can be the corresponding ones of the **type** of **Select** operation used. If you want to know more about this, you may click “[F.7.12 How do I use “ORDER BY” to order the output records of a Select?”](#)”.

Notice that using “**DISTINCT**” makes your Queries slower because the system has to check for duplicate records, and remove them if found, before producing the **output** record-list. If you want to know more, you may click “[K.7.2.1 Why should I use “DISTINCT”, “UNION” and “ORDER BY” only if needed?](#)”.

You can use the “**HAVING**” and/or “**DISTINCT**” and/or “**TOP**” clauses together in a **Select** operation. Remind that if you use “**HAVING**” and/or “**DISTINCT**” plus “**TOP**”, the “**TOP**” clause will be applied **last** (click [F.7.1](#)).

You **cannot** use both the “**DISTINCT**” and “**DISTINCTROW**” clauses in the same **Select** operation.

If you want to **see** the **dataflow** diagram of the **three types** of **Select** operations, you may click “[F.7.3 What is the dataflow of a Select?”](#)”.

If you want to write (syntax) a correct “**WHERE**” clause, you may click [F.7.14](#).

### **F.7.12 How do I use “ORDER BY” to order the output records of a Select?”**

In a **Select** operation (click [F.7.1](#)), the optional “**ORDER BY**” clause indicates the **ordering** of the (distinct) (retained) **modified** records, as follows:

```
ORDER BY      Order_by-exp-1 (order-elements) [DESC]
             [, ...
             , Order_by-exp-w (order-elements) [DESC] ]
```

The order of (distinct) (retained) **modified** records is **unknown** unless you use the optional “**ORDER BY**” clause.

As you may see in the SQL code above, the “**ORDER BY**” keyword is followed by a **list of expressions** “**Order\_by-exp ()**”. These “**ORDER BY**” expressions are built by **combining** some **elements** (called “**order-elements**” above) with functions (excluding SQL aggregate), value operators and constants. The specific “**order-elements**” that can be used to build the “**ORDER BY**” expressions depend on the type of **Select** operation, and on the usage of the “**DISTINCT**” clause, as I explain further below.

Each **output** record is **ordered** on the **result** that its fields produce on the **first** “**ORDER BY**” expression (the leftmost one). In case there are **groups** of **output** records whose field values produce the **same result** in the **first** “**ORDER BY**” expression, then the records in **each such group** are ordered by their **result** from the **second** “**ORDER BY**” expression. In case there are **groups** of **output** records with the **same value** in the **first and second** expressions, then the records in **each such group** are ordered by the value of the **third** expression. As a general rule, in case there are **groups** of **output** records with **same value** in the **first “z”** “**ORDER BY**” expressions, then the records in **each such group** are ordered by the value of the “**z+1**” “**ORDER BY**” expression.

The ordering of the records is done, by default, in **ascending order** (smallest higher up) of the value of each of the “**ORDER BY**” expressions. In case you want to use descending order (largest first) for one (or more) expressions, then you need to add the optional keyword “**DESC**” after “**ORDER BY**” expression that you want. If you want to know more about the way specific data/field types are treated in **ascending/descending order**, you may click “[F.7.12.1 How are the different data/field types ordered by the “ORDER BY” clause?](#)”.

If an “**ORDER BY**” expression produces **Null**, the Query will work normally. Records producing **Null** in one or more “**ORDER BY**” expressions are therefore ordered according to the normal ordering rules, taking into account that **Null** is considered the **lowest** value for any data type (click [F.7.12.1](#)).

If an “**ORDER BY**” expression produces an **exception** (e.g., divide by zero), on most cases the Query will crash.

The **elements** used to write the “**ORDER BY**” expressions depend **firstly** on the usage, or not, of the optional “**DISTINCT**” clause, as follows:

- If you use the “**DISTINCT**” clause, **each** “**ORDER BY**” expression must be **exactly the same** as one of the “**SELECT**” expressions. In other words, **each** “**ORDER BY**” expression must be **exactly the same** as one of “**SELECT**” expressions. As an additional restriction, the “**SELECT**” expressions that are used as “**ORDER BY**” expressions cannot contain any “**Output-field-name**”.

If you want to overcome this overall restriction on the “**ORDER BY**” expressions, you can just remove the “**ORDER BY**” clause from this **Select** operation and enclose it in **another Select** operation (without the “**DISTINCT**” clause), where you can write the “**ORDER BY**” expressions that you want.

If you want to overcome the additional restriction above, you can replace **each** “**Output-field-name**” that is **used** in the “**SELECT**” expression (that you want to use as an “**ORDER BY**” expression) by the “**SELECT**” expression corresponding

to that “**Output-field-name**”.

- If you **do not** use the “**DISTINCT**” clause, each “**ORDER BY**” expression is built combining “**order-elements**” with functions (excluding SQL aggregate), value operators and constants. What “**order-elements**” can be used depends, secondly, on the **type** of **Select** operation (click *F.7.2*), as follows:
  - In a **Select-no\_aggreg**, the “**order-elements**” can **only** be the “**Input-field-names**”.
  - In a **Select-group\_by\_aggreg**, the “**order-elements**” can be:
    - The exact same “**GROUP BY**” expressions
    - **SQL aggregate functions** each having as argument its specific **expression** over the “**Input-field-names**”.

You **cannot** assign an identifier (with an “**AS**” clause) to individual “**GROUP BY**” expressions. Therefore, you have to write them **as such** when you use them as **elements** to build the “**ORDER BY**” expressions. This may be cumbersome when a “**GROUP BY**” expression is large, but there is no way to avoid this.
  - In a **Select-total\_aggreg**, the “**order-elements**” can be **SQL aggregate functions** each having as argument its specific **expression** over the “**Input-field-names**”.

The reason why “**DISTINCT**” has this “strange” effect over “**ORDER BY**” is because using “**DISTINCT**” resembles doing a “**GROUP BY**” over the results of “**SELECT**” expressions. Doing this **aggregates** duplicate records, and therefore, **destroys** all the record’s information (i.e., “**GROUP BY**” expressions as well as **SQL aggregate functions** “**SQL\_agg\_func()**” over expressions over “**Input-field-names**”) **keeping only** the results of the “**SELECT**” expressions. This is why you can **only** “**ORDER BY**” over the “**SELECT**” expressions (that do not use any “**Input-field-names**”). What is not so clear is why you can **only** use the “**SELECT**” expressions **as such**, and you **cannot** use an **expression** built over them.

I want to highlight that whenever there is **any type** of **record aggregation** (i.e., “**GROUP BY**”, “**total\_aggreg**” and/or “**DISTINCT**” record aggregation), the elements you can use in the “**ORDER BY**” expressions are the ones having **the same value** for **all** the records in each **group**.

Using “**ORDER BY**” in nested **Selects** makes your **Query** slower because the system has to order each of the successive **output** record-lists<sup>34</sup>. Since record order is irrelevant during SQL processing, it **only** makes sense to write an “**ORDER BY**” clause in the **outermost Select** operation of a Query. If you want to know more, you may click “*K.7.2.1 Why should I use “DISTINCT”, “UNION” and “ORDER BY” only if needed?*”.

The “**ORDER BY**” clause works the same for the **three types** of **Select** operations

---

<sup>34</sup> The Query optimizer will most likely fix this, but it is a bad practice to write inefficient SQL code and trust that the Query optimizer will fix it.



(click *F.7.2*), although using it in a **Select-total\_aggreg** does not make any sense.

If you want to see the **dataflow** diagram of the **three types of Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.

If you want to write (syntax) a correct “**WHERE**” clause, you may click *F.7.14*.

### **F.7.12.1 How are the different data/field types ordered by the “ORDER BY” clause?**

The way **values** from the different **data/field type** are **ordered** by the “**ORDER BY**” clause is as follows:

- **Null**  
**Null** is **lower** than **any** valid value from **any** data type. Therefore, **Null** is the **lowest** element of all.

If you configure **ascending** (or “**A to Z**”) ordering, all **Null** appear at the top.

- **String** and **Short Text**

**Text strings** are ordered using alphabetical ordering. In alphabetical ordering each character is assigned an **ordering-value**. **Text strings are ordered according to the ordering-value of their first character**. If the first “**n**” characters of two text strings have the same ordering-value, then they are ordered according to the ordering-value of their “**n+1**” character.

**Leading space** characters are treated like any other character. However, **trailing space** characters are **ignored** for ordering purposes. If string **A** is a (case insensitive) prefix of string **B**, then string **A** is lower than string **B**. Consequently, the **zero-length text string** is the **lowest** text string because it is a **prefix** of any other text string (see above).

Any **all-space** string is **also** the **lowest** text string because **trailing space** characters are **ignored** for ordering purposes. Therefore, an all-space string is ordered as if it were the **zero-length** string.

The lowest valued characters are (in increasing value) **single quote** “**”**, **hyphen** “**-**”, **space** “ Chr(32), **no-break space** “ Chr(160), **horizontal tab** “ Chr(9), and **line-feed** “ Chr(10). Notice they are all invisible characters.

The next higher valued characters are “**!**”, “**”**”, “**#**”, “**\$**”, “**%**”, “**&**”, “**(**”, “**)**”, “**\***”, “**+**”, “**,**”, “**.**” and “**/**”, listed in increasing value.

The next higher valued characters are **number** characters “**0**” to “**9**”, in increasing value. Notice that the string “**11**” will be sorted **ahead** of the string “**2**”, and this is different from the resulting sorting if they were numbers.

The next higher valued characters are **letter** characters, having increasing value in alphabetical order. This is, character “**a**” has a lower order-value than character “**b**”, “**b**” has lower value than “**c**” and successively until “**z**”. MS-Access is **case insensitive** for both **comparison operators** and **string ordering**. Therefore, the order-value of a character is the same in upper case and lower case. Each non-English character (“**á**”, “**à**”, “**â**”, ...) has its own specific order-value. Non-English characters have **consecutive** values in respect to their equivalent English character (and therefore, are **sorted consecutively**). Non-English characters have a higher

order-value than their English equivalent, except “ç” that has a lower order-value than “c”.

If you configure **ascending** (or “A to Z”) order, **Null** will appear at the top. Next, **zero-length** strings and **all-space** strings mixed together. Next, strings starting with “”, **hyphen** “-”, **space** “ ” Chr(32), **no-break space** “ ” Chr(160), horizontal **tab** “ ” Chr(9), and **line-feed** “ ” Chr(10), in this order. Next, strings starting with **special** characters. Next, strings starting with **number** characters “0” to “9”. Next, strings starting with **letter** characters in alphabetical order, case insensitive, “a/A” to “z/Z”. Strings starting with non-English letters go lower than the strings starting with the equivalent English letter (with some exceptions). An example of exceptions is that strings starting with “ç” go higher than strings starting with “c”.

Remind that the string “11” will be sorted higher than string “2”, and this is different from the resulting sorting if they were numbers.

- **Date** and **Date/Time**

Are ordered according to their equivalent numeric value (click *D.4.5*).

If you configure **ascending** order, **Null** will appear at the top, followed by oldest date-time, and most future date-time will appear at the bottom.

- **Boolean** and **Yes/No**

**True/Yes/On** or **ticked** is lower than **False/No/Off** or **unticked**, according to their equivalent numeric value (click *D.4.7*). If you configure ascending order, **Null** will appear at the top, followed by **True/Yes/On** or **ticked**, followed by **False/No/Off** or **unticked** that will appear at the bottom.

- **Integer, Long, Single, Double, Currency** and **Integer, Long Integer, Single, Double, Currency**

Are ordered according to their numeric value. If you configure ascending order, **Null** will appear at the top, followed by the most negative value (negative value with highest absolute value) and the largest positive value will appear at the bottom.

### F.7.13 What is the “TOP” clause of a Select?

In a **Select** operation (click *F.7.1*), the optional “**TOP**” clause indicates what are the **top output** records (discarding the others), as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
```

The optional “**TOP int**” clause **retains** the **first “int+” output** records, according to the “**ORDER BY**” ordering criterion, while the others are **discarded**.

If you add the optional “**PERCENT**” keyword, then “**TOP int PERCENT**” **retains** the **first “int+” percent** records, according to the “**ORDER BY**” ordering criterion. Notice that records whose percentage is fractional, but **lower** than “**int**” **percent** will be **retained**. For example, if you have **ten** ordered records, and you indicate “**1 PERCENT**”, this will retain **one** record (the first one). If you rather indicate “**11 PERCENT**”, this will retain **two** records.

Notice that I indicate “**int+**” and not “**int**” because **more** than “**int**”, or “**int**” percent, records will be produced in case there is a **tie** according to the “**ORDER BY**” criterion. If according to the “**ORDER BY**” criterion there is a **tie** between the “**int<sup>th</sup>**”, or the “**int<sup>th</sup>**”

**percent**", record and **one or more** subsequent records, **all the tied records** will be produced.

The parameter "**int**" **must** be a **positive integer** constant: if "**int**" is an expression or a zero or negative constant, it is a syntax error and you will not be able to save the Query. For the case of "**TOP int PERCENT**", the parameter "**int**" must be an integer constant lesser than or equal 100: if it is greater than 100, it is a syntax error and you will not be able to save the Query.

If the parameter "**int**" is a **fractional** constant, MS-Access will **remove** the period ".", and consider the result an **integer** number. For example, if you write "**0.5**", it will be considered "**5**"; if you write "**0.50**", it will be considered "**50**"; if you write "**1.2**", it will be considered "**12**". This may be an MS-Access **bug**.

You can use the "**HAVING**" and/or "**DISTINCT**" and/or "**TOP**" clauses together in a **Select** operation. Remind that if you use "**HAVING**" and/or "**DISTINCT**" plus "**TOP**", the "**TOP**" clause will be applied **last** (click *F.7.1*).

You **can** use the "**TOP**" clause **without** the "**ORDER BY**" clause. However, this does **not make much sense** because the record order would be unknown, and you would basically get the "**TOP**" records from an unknown order.

You **can** use the "**TOP**" clause in a **Select-total\_aggreg**. However, this does **not make much sense** because a **Select-total\_aggreg** only produce **one** (or none) record, to getting the top out of one does not seem very useful.

Very important to highlight that if the **input** record-list is a Table name, no records will be **deleted** from the Table, and the Table remains **unmodified**. This is so because all the SQL **consulting** operators work over an **image** of Table's records, and **not** over the Table records themselves. Remind that the **Select** operator is a **consulting** SQL operator (i.e., one that can only consult Tables) and it is not a **data-changing** SQL operator. If you want to actually **modify** your Table's records, you may click "*F.6.2 What are the SQL data-changing operators?*".

If you want to **see** the **dataflow** diagram of the **three types** of **Select** operations, you may click "*F.7.3 What is the dataflow of a Select?*".

If you want to write (syntax) a correct "**WHERE**" clause, you may click *F.7.14*.

### **F.7.14 How do I write a correct (syntax) Select?**

For all **Select** operations it is possible to use "**Output-field-names**" in the "**SELECT**" **expressions** "**Output-exp()**" **1** to **n** (as long as you **do not** create a **circular** reference, **nor** a **nested** SQL aggregate function, see below).

Using an "**Output-field-name**" in this way allows to **reuse** the expression from **one output** field to write the "**SELECT**" **expression** of **another output** field.

Creating a **circular** reference is when you are assigning an **output** field name to **itself**, or when you are assigning **output** field names in a circular way (e.g., "**Name\_1**" is used in the expression for "**Name\_2**", "**Name\_2**" is used in the expression for "**Name\_3**" and "**Name\_3**" is used in the expression for "**Name\_1**").

Creating a nested SQL aggregate function is when the argument of an SQL aggregate function contains another SQL aggregate function. For example, the following

“**SELECT**” **expressions** creates a **nested SQL** aggregate function:

```
SELECT Sum(Rainfall) AS Rain, Avg(Rain+Humid) AS Combined
```

Since the **output field name** “**Rain**” represents “**Sum(Rainfall)**”, where “**Sum()**” is an SQL aggregate function, using “**Rain**” inside the SQL aggregate function “**Avg()**” creates a nested SQL aggregate function.

Notice also that “**Output-field-names**” **cannot** be used in the “**HAVING**” **expression**, nor in the “**ORDER BY**” **expressions**.

The outermost **Select** operation of a Query may be enclosed between parentheses.

Depending on what **type** (click *F.7.2*) of **Select operation** you want to write, you may click:

- “*F.7.15 How do I write a correct (syntax) Select-no\_aggreg?*”
- “*F.7.16 How do I write a correct (syntax) Select-group\_by\_aggreg?*”
- “*F.7.17 How do I write a correct (syntax) Select-total\_aggreg?*”

### How do I nest Selects?

The **Select** operations can be nested, so you may write a **Select** operation as the **input** record-list of another **Select** operation. Each **Select** operation that is an **input** record-list **must be enclosed in parentheses**. The following nested **Select** operation<sup>35</sup> is correct:

```
SELECT City AS Non_capital_cities
FROM
(
  SELECT City
  FROM
    (SELECT City
     FROM T_Subsiary_sites
     WHERE City <> "Washington")
   WHERE City <> "Paris"
)
WHERE City <> "Madrid"
```

Notice I have colored matching parentheses for your convenience.

### F.7.15 How do I write a correct (syntax) Select-no\_aggreg?

A **Select-no\_aggreg** is when you **do not** use the “**GROUP BY**” clause (click *F.7.9*), nor any **SQL aggregate function** (click *F.7.18*). You may click:

- “*F.7.15.1 What is a syntax-example of a Select-no\_aggreg?*”
- “*F.7.15.2 What are the formal rules (syntax) to write a Select-no\_aggreg?*”

<sup>35</sup> This is the Query “**F\_Select\_nested**” in file “**Company\_Database.accdb**”.

### F.7.15.1 What is a syntax-example of a Select-no aggreg?

An illustrative example of a fairly complete **Select-no\_aggreg**<sup>36</sup> is:

```
SELECT TOP 80 PERCENT "City_" & Capital AS Out_fld
, Len(Out_fld) + Temp_Min AS Nonsense
FROM T_Capital_Temps
WHERE Capital <> "Beijing"
ORDER BY 5*Temp_Max, Exp(Temp_Min) DESC
```

The optional “**DISTINCT**” clause is **not** used, to allow using more flexible “**ORDER BY**” expressions.

The optional “**TOP**” clause is used, in its “**PERCENT**” variant.

There are **two** “**SELECT**” expressions:

- The **two** “**SELECT**” expressions are assigned an **output field name** using the optional “**AS**” clause. The **two field names** are “**Out\_fld**” and “**Nonsense**”.
- The “**SELECT**” expression for **output field** “**Out\_fld**” uses as an **element** the **input field name** “**Capital**”.
- The “**SELECT**” expression for **output field** “**Nonsense**” uses as **elements** the **output field name** “**Out\_fld**” and the **input field name** “**Temp\_Min**”.

The **input** record-list in the “**FROM**” clause is:

- The Table name “**T\_Capital\_temps**” (click *F.10.5*).

The **elements** of the “**WHERE**” *Boolean-expression* are:

- The **input field name** “**Capital**”.

The **two** “**ORDER BY**” expressions are:

- “**5\*Temp\_Max**” in ascending (default) order. This expression uses as an **element** the **input field name** “**Temp\_Max**”.
- “**Exp(Temp\_Min)**” using the keyword “**DESC**” to indicate descending order. This expression uses as an **element** the **input field name** “**Temp\_Min**”.
- Notice how the “**ORDER BY**” expressions may be totally unrelated to the “**SELECT**” expressions.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “*F.11.2 What are the SQL color codes used in this Guide?*”.

---

<sup>36</sup> This is the Query “**F\_Select\_w\_no\_aggreg\_Syn**” from the “**Company\_Database.accdb**” file.

### F.7.15.2 What are the formal rules (syntax) to write a **Select-no aggreg**?

A correct **Select-no aggreg** has to be written in the following way:

```

SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
    {
        * or
        Output-exp_1(INOUT-field-names) [AS Output-field-name_1]
    [ , ...
        , Output-exp_n(INOUT-field-names) [AS Output-field-name_n] ] }

[ FROM { [[ (] {Table-name or Query-name} [)]
or [ {Table-name or Query-name} [AS Input-record-list-name] ]
or [ ( {Select-opr or Union-opr } ) [AS Input-record-list-name] ]
or [[ (] Inner-or-Outer-Join-opr [)] or Cross-Join-opr [)] ] ]

[ WHERE Where-Boolean-exp (Input-field-names) ]

[ ORDER BY Order_by-exp-1 () [DESC]
    [ , ...
        , Order_by-exp-w () [DESC] ] ]

```

The words in **bold** font are SQL **keywords**. The elements enclosed in square brackets “[ ]” are optional, and you may use each of these elements or not, depending on your desired result from the **Select** operation. The elements separated with “or” are alternative options. Each list of alternative options for an element is enclosed between curly braces “{ }” when the element is **not optional** and between square brackets “[ ]” when the element is **optional**. Some curly braces “{ }” and square brackets “[ ]” are colored just to make it easier to see which ones are paired.

The elements in gray text are the ones that I advise you do not use:

- I advise you **do not use** the optional “**DISTINCTROW**” clause because it is not compatible with SQL server, and because you can get its same functionality in an easy way. If you want to know more, you may click “[F.7.8 What is the “DISTINCTROW” clause of a Select?](#)”.
- When the list of “**SELECT**” expressions “**Output-exps ()**” is exactly the same as the list of **all input field names**, you can replace the list of “**SELECT**” expressions by just one “\*”. My advice is you **do not** use an “\*”, because it is less readable than the explicit list of **input field names** and also because it can create some problems (click [K.4.9](#)).
- I advise you **do not use** the optional “**ALL**” clause because it does not have any effect.

All the terms above with a trailing “-exp” are **expressions**. The **terms** enclosed between the **parentheses** of each expression are the **elements** that can be used to build that specific expression, by **combining** these **elements** with functions (**excluding** SQL aggregate), value operators and constants.

All the terms above with a trailing “-opr” are SQL operations.

The “**Input-field-names**” are the **1** to **v** **input** field names from the **input** record-list.

The “**INOUT-field-names**” are the **1** to **v** “**Input-field-names**” and/or the **1** to **n** “**Output-field-names**”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click

*“F.11.2 What are the SQL color codes used in this Guide?”*.

I will now explain how to write the different clauses of the **Select-no\_agg** in the order they appear in the SQL code, because I think this is more convenient for syntax purposes.

### How do I write the “SELECT” clause?

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
{ * or
  Output-exp_1(INOUT-field-names) [AS Output-field-name_1]
  [, ...
  , Output-exp_n(INOUT-field-names) [AS Output-field-name_n] ]}
```

After the “**SELECT**” keyword (and after the optional clauses “**DISTINCT**”, “**TOP**” and/or “**DISTINCTROW**”, in case you are using them) you write the “**SELECT**” **expressions** “**Output-exp()**” separated with commas “,”. **Each** “**SELECT**” **expression** that you write produces **one output field**.

Each “**Output-exp()**” is built over the “**Input-field-names**” and/or over the “**Output-field-names**” (as long as you **do not** create a **circular** reference, click *F.7.14*).

Although you can use “**Output-field-names**” in the “**SELECT**” **expressions**, remind you **cannot** use them in the “**ORDER BY**” expressions.

If you add after a “**SELECT**” **expression** the keyword “**AS**” followed by a **name**, this will be the **name** of this **output field**. If you do **not add** this “**AS**” clause, then you have two cases:

- If the “**Output-exp\_i()**” consists **exactly** of a single **input field name**, then the **output field name** will be that **input field name**.
- Otherwise, MS-Access will assign to that “**Output-exp\_i()**” the **output field name** “**ExprXXXX**”, where “**XXXX**” is a four-digit integer number.

If you want to know more about the “**SELECT**” clause, you may click “*F.7.5 What are the output fields (“SELECT” clause) of a Select?”*”.

### How do I write the “DISTINCT”, “TOP” and “DISTINCTROW” clauses?

The optional “**DISTINCT**” clause is written by adding the “**DISTINCT**” keyword after the “**SELECT**” keyword.

If you want to know more about the “**DISTINCT**” clause, you may click “*F.7.11 What is the “DISTINCT” clause of a Select?”*”.

The optional “**TOP**” clause is written by adding “**TOP int**” after the “**SELECT**” keyword. If you use the **percent** variant, then it is written by adding “**TOP int PERCENT**” after the “**SELECT**” keyword. In case “**DISTINCT**” is used, then the “**TOP**” clause is written after the “**DISTINCT**” keyword. The text “**int**” stands for a positive integer constant. For the “**PERCENT**” variant, “**int**” cannot be greater than 100.

If you want to know more about the “**TOP**” clause, you may click “*F.7.13 What is the “TOP” clause of a Select?”*”.

I advise you do not use the optional “**DISTINCTROW**” clause. If in spite of my advice



you want to use it, you just write the keyword “**DISTINCTROW**” after the “**SELECT**” keyword (or after the “**TOP**” clause, in case it is used). Remind that you cannot use **both** “**DISTINCT**” and “**DISTINCTROW**” in the same **Select** operation.

If you want to know more about the “**DISTINCTROW**” clause, you may click “[F.7.8 What is the “DISTINCTROW” clause of a Select?](#)”.

### How do I write the “**FROM**” clause?

```
FROM { [( {Table-name or Query-name} )]
      or [ {Table-name or Query-name} [AS Input-record-list-name] ]
      or [ ( {Select-opr or Union-opr } ) [AS Input-record-list-name] ]
      or [( { Inner-or-Outer-Join-opr } ) or Cross-Join-opr ] }
```

Right after the “**FROM**” keyword you write the **input record-list**. The **input** record-list can be a Table name or a Query name or a **Join** operation or a **Select** operation or a **Union** operation. Notice this is **different** from the writing rules (syntax) of the **Join** operation.

Let me clarify the rules for **parentheses** and the “**AS**” clause, depending on what is the **input record-list** after the “**FROM**” keyword:

- **A Table name or Query name**  
It **may** be enclosed between parentheses, or, it **may** have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one of the **Join** operation.
- **A Select operation or Union operation**  
It **must** be enclosed between parentheses. It **may also** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the one of the **Join** and **Union** operations.
- **A Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.
- **A Cross-Join operation**  
It **must not** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.

If you want to know more about the “**FROM**” clause, you may click “[F.7.4 What is the input record-list \(“FROM” clause\) of a Select?](#)”.

### How do I write the “**WHERE**” clause?

```
WHERE Where-Boolean-exp (Input-field-names)
```

The “**Where-Boolean-exp ( )**” **expression** that you write after the “**WHERE**” keyword is built by combining the **input field names** with functions (**excluding** SQL aggregate), value operators and constants.

If you want to know more about the “**WHERE**” clause, you may click “[F.7.7 What is the “WHERE” clause of a Select?](#)”.



### How do I write the “ORDER BY” clause?

Each of the expressions “`Order_by-exp-i ()`” that you write after the “ORDER BY” keyword depends on the usage, or not, of the optional “DISTINCT” clause, as follows:

If you use the “DISTINCT” clause,

then each expression used in the “ORDER BY” clause **must be exactly the same** as one of the “SELECT” expressions. The resulting “ORDER BY” clause would then be:

```
ORDER BY   Output-exp_x()-not-having-Output-field-names [DESC]
          [, ...
          , Output-exp_y()-not-having-Output-field-names [DESC] ]
```

Moreover, you can **only** use the “SELECT” expressions “`Output-exp_i ()`” that **do not** contain any “`Output-field-name`”.

If you **do not** use the “DISTINCT” clause,

then each “`Order_by-exp-i ()`” “ORDER BY” expression is built over the “`Input-field-names`”. The resulting “ORDER BY” clause would then be:

```
ORDER BY   Order_by-exp_1(Input-field-names) [DESC]
          [, ...
          , Order_by-exp_w(Input-field-names) [DESC] ]
```

If you want to know more about the “ORDER BY” clause, you may click “[F.7.12 How do I use “ORDER BY” to order the output records of a Select?](#)”.

### Final remarks

The outermost `Select-no_aggreg` operation of a Query may be enclosed between parentheses.

You probably have noticed that the “FROM” clause at the beginning of this section is enclosed between square brackets, which implies that the “FROM” clause is **optional**. This is not a typo, and it is actually correct. The following `Select` operation<sup>37</sup> **without** a “FROM” clause is valid:

```
SELECT "New York" AS City, "Manhattan" AS District ;
```

This operation produces **one** record with the constant field values indicated in the “SELECT” expressions. Although this `Select` operation is correct, producing only **one** record with **constant** field values is of very limited use.

Finally, if you want to **see** the **dataflow** diagram of the **three types** of `Select` operations, you may click “[F.7.3 What is the dataflow of a Select?](#)”.

### F.7.16 How do I write a correct (syntax) Select-group by aggreg?

A `Select-group_by_aggreg` is when you use the “GROUP BY” clause (click [F.7.9](#)). You may click:

- “[F.7.16.1 What is a syntax-example of a Select-group by aggreg?](#)”
- “[F.7.16.2 What are the formal rules \(syntax\) to write a Select-group by aggreg?](#)”

---

<sup>37</sup> This is the Query “`F_Select_without_FROM`” from file “`Company_Database.accdb`”.

### F.7.16.1 What is a syntax-example of a Select-group by aggreg?

An illustrative example of a fairly complete Select-group\_by\_aggreg<sup>38</sup> is:

```
SELECT TOP 80 PERCENT
    Len("City_" & Capital)                AS Out_fld
    , 2 * Len(Out_fld)                    AS Nonsense_2
    , 3 * Sum(3 + Temp_Max)                AS Nonsense_3
    , 4 * (1+(9+Temp_min) + Max(5*Temp_Max)) AS Nonsense_4
    , 5 * Avg(Len(Out_fld))                AS Nonsense_5
FROM T_Capital_Temps
WHERE Capital <> "Beijing"
GROUP BY "City_" & Capital, 9+Temp_Min
HAVING (Len("City_" & Capital) < 2) OR (Min(3*Temp_Min) < 0)
ORDER BY 5*(9+Temp_Min) DESC, 7+StDev(Temp_Min+Temp_Max)
```

The optional “**DISTINCT**” clause is **not** used, to allow using more flexible “**ORDER BY**” expressions.

The optional “**TOP**” clause is used, in its “**PERCENT**” variant.

There are **five** “**SELECT**” expressions:

- The **five** “**SELECT**” expressions are assigned an **output field name** using the optional “**AS**” clause. The **five output field names** are “**Out\_fld**”, “**Nonsense\_2**”, “**Nonsense\_2**”, “**Nonsense\_3**”, “**Nonsense\_4**” and “**Nonsense\_5**”.
- The “**SELECT**” expression for output field “**Out\_fld**” uses as an **element** the first “**GROUP BY**” expression “**"City\_" & Capital**”.
- The “**SELECT**” expression for output field “**Nonsense\_2**” uses as an **element** the **output field name** “**Out\_fld**”.
- The “**SELECT**” expression for output field “**Nonsense\_3**” uses as an **element** the SQL aggregate function “**Sum()**” over an **expression** that uses as an **element** the **input field name** “**Temp\_Max**”.
- The “**SELECT**” expression for output field “**Nonsense\_4**” uses as an **element** the second “**GROUP BY**” expression “**9+Temp\_min**”. It also uses as another **element** the SQL aggregate function “**Max()**” over an **expression** that uses as an **element** the **input field name** “**Temp\_Max**”.
- The “**SELECT**” expression for output field “**Nonsense\_5**” uses as an **element** the SQL aggregate function “**Avg()**” over an **expression** that uses as an **element** the **output field name** “**Out\_fld**”.

The **input** record-list in the “**FROM**” clause is:

- The Table name “**T\_Capital\_temps**” (click *F.10.5*).

The **elements** of the “**WHERE**” *Boolean-expression* are:

- The **input field name** “**Capital**”.

<sup>38</sup> This is the Query “**F\_Select\_w\_group\_by\_aggreg\_Syn**” from the “**Company\_Database.accdb**” file.

The two “GROUP BY” expressions are:

- “**City\_** & **Capital**”, which uses as an element the input field name “**Capital**”.
- “**9+Temp\_Min**”, which uses as an element the input field name “**Temp\_Min**”.

The elements of the “HAVING” Boolean-expression are:

- The first “GROUP BY” expression “**9+Temp\_min**”.
- The SQL aggregate function “**Min()**” over an expression that uses as an element the input field name “**Temp\_Min**”.

The two “ORDER BY” expressions are:

- “**5\*(9+Temp\_Min)**” using the keyword “**DESC**” to indicate descending order. This expression uses as an element the second “GROUP BY” expression “**9+Temp\_min**”.
- “**7+StDev(Temp\_Min+Temp\_Max)**” in ascending (default) order. This expression uses as an element the SQL aggregate function “**StDev()**” over an expression that uses as elements the two input field names “**Temp\_Min**” and “**Temp\_Max**”.
- Notice how the “ORDER BY” expressions may be totally unrelated to the “SELECT” expressions.

Let me point out a few relevant issues:

- If you remove the parentheses enclosing the “GROUP BY” expression “**9+Temp\_min**” from the fourth “SELECT” expression, the Query will produce a syntax error. This is because without the parentheses, the evaluation order of the expression evaluates first “**1+9**”, and this causes that the “GROUP BY” expression is not recognized.
- If you add the “SELECT” expression “**Count(Nonsense\_3)**”, the Query will produce a syntax error. This is because you are using the “**Sum()**” SQL aggregate function as an argument of the “**Count()**” SQL aggregate function, through the output field name “**Nonsense\_3**”. You are then nesting SQL aggregate functions, which is not allowed.

If you want to know the SQL color codes used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

### F.7.16.2 What are the formal rules (syntax) to write a Select-group by aggreg?

A correct **Select-group\_by\_aggreg** has to be written in the following way:

```

SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , Group_by-exp_1(Input-field-names)
                  , ...
                  , Group_by-exp_k(Input-field-names)
                  , SQL_agg_func(exp_o11(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_o1y(INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
    [, ...
      , Output-exp_n( Output-field-names
                  , Group_by-exp_1(Input-field-names)
                  , ...
                  , Group_by-exp_k(Input-field-names)
                  , SQL_agg_func(exp_on1(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_onz(INOUT-field-names))
                  ) [ AS Output-field-name_n ] ]

FROM { [( {Table-name or Query-name} )]
        or [ {Table-name or Query-name} [AS Input-record-list-name] ]
        or [ ( {Select-opr or Union-opr} ) [AS Input-record-list-name] ]
        or [( { Inner-or-Outer-Join-opr } ) or Cross-Join-opr ] ]

[ WHERE Where-Boolean-exp(Input-field-names) ]

GROUP BY      Group_by-exp_1(Input-field-names)
              [ , ...
              , Group_by-exp_k(Input-field-names) ]

[ HAVING Having-Boolean-exp( Group_by-exp_1(Input-field-names)
                              , ...
                              , Group_by-exp_k(Input-field-names)
                              , SQL_agg_func(exp_h1(Input-field-names))
                              , ...
                              , SQL_agg_func(exp_ht(Input-field-names)) ) ]

[ ORDER BY      Order_by-exp-1() [DESC]
              [ , ...
              , Order_by-exp-w() [DESC] ] ]

```

The words in **bold** font are SQL **keywords**. The elements enclosed in square brackets “[ ]” are optional, and you may use each of these elements or not, depending on your desired result from the **Select** operation. The elements separated with “or” are alternative options. Each list of alternative options for an element is enclosed between curly braces “{ }” when the element is **not optional** and between square brackets “[ ]” when the element is **optional**. Some curly braces “{ }” and square brackets “[ ]” are colored just to make it easier to see which ones are paired.

The elements in gray text are the ones I advise you do not use:

- I advise you **do not use** the optional “**DISTINCTROW**” clause because it is not compatible with SQL server, and because you can get its same functionality in an easy way. If you want to know more, you may click “[F.7.8 What is the “DISTINCTROW” clause of a Select?](#)”.
- I advise you **do not use** the optional “**ALL**” clause because it does not have any

effect.

All the terms above with a trailing “-exp” are **expressions**. The **terms** enclosed between the **parentheses** of each expression are the **elements** that can be used to build that specific expression, by **combining** these **elements** with functions (**excluding** SQL aggregate), value operators and constants.

All the terms above with a trailing “-opr” are SQL operations.

The “**Input-field-names**” are **all** the **input** field names from the **input** record-list.

The “**INOUT-field-names**” are the **1** to **v** “**Input-field-names**” and/or the **1** to **n** “**Output-field-names**”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

I will now explain how to write the different clauses of the **Select-group\_by\_aggreg** in the order they appear in the SQL code, because I think this is more convenient for syntax purposes.

### How do I write the “SELECT” clause?

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , Group_by-exp_1(Input-field-names)
                  , ...
                  , Group_by-exp_k(Input-field-names)
                  , SQL_agg_func(exp_o11(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_o1y(INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
[ , ...
  , Output-exp_n( Output-field-names
                  , Group_by-exp_1(Input-field-names)
                  , ...
                  , Group_by-exp_k(Input-field-names)
                  , SQL_agg_func(exp_on1(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_onz(INOUT-field-names))
                  ) [ AS Output-field-name_n ] ]
```

After the “**SELECT**” keyword (and after the optional clauses “**DISTINCT**”, “**TOP**” and/or “**DISTINCTROW**”, in case you are using them) you write the “**SELECT**” **expressions** “**Output-exp()**” separated with commas “**,**”. **Each** “**SELECT**” **expression** that you write produces **one output field**.

Each “**Output-exp()**” is built over the following **elements**:

- “**Output-field-names**” **1** to **n**  
This is, **other output field names** (as long as you **do not** create a **circular** reference, click [F.7.14](#)).
- “**Group\_by-exp(Input-field-names)**” **1** to **k**  
This is, any number of **exactly the same** “**GROUP BY**” **expressions** (click [F.7.9](#)).
- “**SQL\_agg\_func(exp(INOUT-field-names))**”  
This is, any number of **SQL aggregate functions** (click [F.7.18](#)) each having as argument its specific **expression** over the “**Input-field-names**” and/or over

the “**Output-field-names**” (as long as you **do not** create a **circular** reference, **nor** a **nested** SQL aggregate function, click *F.7.14*).

Although you can use “**Output-field-names**” in the “**SELECT**” **expressions**, remind that you **cannot** use them in the “**HAVING**” expressions, nor in the “**ORDER BY**” expressions.

You **cannot** assign an identifier (with an “**AS**” clause) to individual “**GROUP BY**” **expressions**. Therefore, you have to write them **as such** when you use them as **elements** to build the “**SELECT**” **expressions**. This may be cumbersome when a “**GROUP BY**” **expression** is large, but there is no way to avoid this.

When you use the “**GROUP BY**” **expressions** as **part** of a larger “**SELECT**” **expression**, I strongly advice that you write **each** of them between **parentheses** to prevent that MS-Access rejects the larger “**SELECT**” **expression** due to a different **evaluation order** than the one you expected (see the example in *F.7.16.1*).

If you add after a “**SELECT**” **expression** the keyword “**AS**” followed by a **name**, this will be the **name** of this **output** field. If you do **not add** this “**AS**” clause, then you have two cases:

- If the “**Output-exp\_i ()**” consists **exactly** of a single **input field name**, then the **output field name** will be that **input field name**.
- Otherwise, MS-Access will assign to that “**Output-exp\_i ()**” the **output field name** “**ExprXXXX**”, where “**XXXX**” is a four-digit integer number.

Be aware that an expression “**exp\_o ()**” that is the argument of a given SQL aggregate function **cannot** contain any SQL aggregate function. This **cannot** happen in a direct manner, **nor through a reference** to another “**Output-field-name**”.

If you want to know more about the “**SELECT**” clause, you may click “*F.7.5 What are the output fields (‘SELECT’ clause) of a Select?*”.

### How do I write the “**DISTINCT**”, “**TOP**” and “**DISTINCTROW**” clauses?

The optional “**DISTINCT**” clause is written by adding the “**DISTINCT**” keyword after the “**SELECT**” keyword.

If you want to know more about the “**DISTINCT**” clause, you may click “*F.7.11 What is the ‘DISTINCT’ clause of a Select?*”.

The optional “**TOP**” clause is written by adding the “**TOP int**” after the “**SELECT**” keyword. If you use the **percent** variant, then it is written by adding “**TOP int PERCENT**” after the “**SELECT**” keyword. In case “**DISTINCT**” is used, then the “**TOP**” clause is written after the “**DISTINCT**” keyword. The symbol “**int**” stands for a positive integer constant, not greater than 100 for the **percent** variant.

If you want to know more about the “**TOP**” clause, you may click “*F.7.13 What is the ‘TOP’ clause of a Select?*”.

I advise you do not use the optional “**DISTINCTROW**” clause. If in spite of my advice you want to use it, you just write the keyword “**DISTINCTROW**” after the “**SELECT**” keyword (or after the “**TOP**” clause, in case it is used). Remind that you cannot use **both** “**DISTINCT**” and “**DISTINCTROW**” in the same **Select** operation.



If you want to know more about the “**DISTINCTROW**” clause, you may click [“F.7.8 What is the “\*DISTINCTROW\*” clause of a \*Select\*?”](#).

### How do I write the “**FROM**” clause?

```
FROM { [[ ( {Table-name or Query-name} ) ] ]
      or [ {Table-name or Query-name} [AS Input-record-list-name] ]
      or [ ( {Select-opr or Union-opr } ) [AS Input-record-list-name] ]
      or [ [ ( Inner-or-Outer-Join-opr ) ] or Cross-Join-opr ] }
```

Right after the “**FROM**” keyword you write the **input record-list**. The **input** record-list can be a Table name or Query name, or a **Join** operation, **Select** operation or **Union** operation. Notice this is **different** from the writing rules (syntax) of the **Join** operation.

Let me clarify the rules for **parentheses** and the “**AS**” clause, depending on what is the **input record-list** after the “**FROM**” keyword:

- **A Table name or Query name**  
It **may** be enclosed between parentheses, or, it **may** have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one of the **Join** operation.
- **A Select operation or Union operation**  
It **must** be enclosed between parentheses. It **may also** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the one of the **Join** and **Union** operations.
- **A Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.
- **A Cross-Join operation**  
It **must not** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.

If you want to know more about the “**FROM**” clause, you may click [“F.7.4 What is the \*input record-list\* \(“\*FROM\*” clause\) of a \*Select\*?”](#).

### How do I write the “**WHERE**” clause?

```
WHERE Where-Boolean-exp (Input-field-names)
```

The “**Where-Boolean-exp ( )**” expression that you write after the “**WHERE**” keyword is built by combining the **input field names** with functions (**excluding** SQL aggregate), value operators and constants.

If you want to know more about the “**WHERE**” clause, you may click [“F.7.7 What is the “\*WHERE\*” clause of a \*Select\*?”](#).

### How do I write the “**GROUP BY**” clause?

```
GROUP BY Group_by-exp_1 (Input-field-names)
         [ , ...
         , Group_by-exp_k (Input-field-names) ]
```

Each of the **expressions** “**Group\_by-exp-i (Input-field-names)**” that you write after the “**GROUP BY**” keyword is built by combining the **input field names** with

functions (**excluding** SQL aggregate), value operators and constants.

If you want to know more about the “**GROUP BY**” clause, you may click “[F.7.9 What is the “GROUP BY” clause of a Select-group by aggreg?](#)”.

### How do I write the “**HAVING**” clause?

```
HAVING Having-Boolean-exp( Group_by-exp_1(Input-field-names)
, ...
, Group_by-exp_k(Input-field-names)
SQL_agg_func(exp_h1(Input-field-names))
, ...
, SQL_agg_func(exp_ht(Input-field-names)) )
```

The “**Having-Boolean-exp()**” expression in the “**HAVING**” clause is built over the elements:

- “**Group\_by-exp(Input-field-names)**” 1 to k  
This is, any number of **exactly the same** “**GROUP BY**” **expressions** (click [F.7.9](#)).
- “**SQL\_agg\_func(exp(Input-field-names))**”  
This is, any number of **SQL aggregate functions** (click [F.7.18](#)) each having as argument its specific **expression** over the “**Input-field-names**”.

Notice that these are **the same elements** you can use for the expressions in the “**ORDER BY**” clause when “**DISTINCT**” is not used.

You **cannot** assign an identifier (with an “**AS**” clause) to individual “**GROUP BY**” **expressions**. Therefore, you have to write them **as such** when you use them as **elements** to build the “**HAVING**” **expression**. This may be cumbersome when a “**GROUP BY**” **expression** is large, but there is no way to avoid this.

When you use the “**GROUP BY**” **expressions** as **part** of a larger “**HAVING**” **expression**, I strongly advise you write **each** of them between **parentheses** to prevent that MS-Access rejects the larger “**HAVING**” **expression** due to a different **evaluation order** than the one you expected (see the example in [F.7.16.1](#)).

If you want to know more about the “**HAVING**” clause, you may click “[F.7.10 What is the “HAVING” clause of Select-group by aggreg or Select-total aggreg?](#)”.

### How do I write the “**ORDER BY**” clause?

Each of the expressions “**Order\_by-exp-i()**” that you write after the “**ORDER BY**” keyword depends on the usage, or not, of the optional “**DISTINCT**” clause, as follows:

If you **use** the “**DISTINCT**” clause,

then each expression used in the “**ORDER BY**” clause **must be exactly the same** as one of the “**SELECT**” **expressions**. Moreover, you can **only** use the “**SELECT**” **expressions** “**Output-exp\_i()**” that **do not** contain any “**Output-field-name**”. The resulting “**ORDER BY**” clause would then be:

```
ORDER BY Output-exp-r()-not-having-Output-field-names [DESC]
[ , ...
, Output-exp-t()-not-having-Output-field-names [DESC] ]
```

If you **do not** use the “**DISTINCT**” clause,

then **each** “**Order\_by-exp-i()**” expression in the “**ORDER BY**” list is built over the



elements:

- **Group\_by-exp (Input-field-names) 1 to k**  
This is, any number of **exactly the same** “GROUP BY” **expressions** (click *F.7.9*).
- **SQL\_agg\_func (exp (Input-field-names) )**  
This is, any number of SQL aggregate functions (click *F.7.18*) each having as argument its specific **expression** over the “Input-field-names”.

Notice these are **the same elements** you can use for the “HAVING” *Boolean*-expression.

The resulting “ORDER BY” clause would then be:

```
ORDER BY Order_by-exp-1( Group_by-exp_1 (Input-field-names)
                        / ...
                        / Group_by-exp_k (Input-field-names)
                        / SQL_agg_func (exp_b11 (Input-field-names))
                        / ...
                        / SQL_agg_func (exp_b1q (Input-field-names))
                        ) [DESC]
[ , ...
  , Order_by-exp-w( Group_by-exp_1 (Input-field-names)
                  / ...
                  / Group_by-exp_k (Input-field-names)
                  / SQL_agg_func (exp_bw1 (Input-field-names))
                  / ...
                  / SQL_agg_func (exp_bwm (Input-field-names))
                  ) [DESC] ]
```

You **cannot** assign an identifier (with an “AS” clause) to individual “GROUP BY” **expressions**. Therefore, you have to write them **as such** when you use them as **elements** to build the “ORDER BY” **expressions**. This may be cumbersome when a “GROUP BY” **expression** is large, but there is no way to avoid this.

When you use the “GROUP BY” **expressions** as **part** of a larger “ORDER BY” **expression**, I strongly advise you write **each** of them between **parentheses** to prevent that MS-Access rejects the **larger** “ORDER BY” **expression** due to a different **evaluation order** than the one you expected (see the example in *F.7.16.1*).

If you want to know more about the “ORDER BY” clause, you may click “*F.7.12 How do I use “ORDER BY” to order the output records of a Select?*”.

If you want to **see** the **dataflow** diagram of the **three types** of **Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.

### Final remarks

The outermost **Select-group\_by\_aggreg** operation of a Query may be enclosed between parentheses.

Finally, if you want to **see** the **dataflow** diagram of the **three types** of **Select** operations, you may click “*F.7.3 What is the dataflow of a Select?*”.

### F.7.17 How do I write a correct (syntax) Select-total aggreg?

A **Select-total\_aggreg** is when you **do not** use the “GROUP BY” clause (click *F.7.9*), **but you use SQL aggregate functions** (click *F.7.18*). You may click:

- “*F.7.17.1 What is a syntax-example of a Select-total\_aggreg?*”

- “F.7.17.2 What are the formal rules (syntax) to write a Select-total aggreg?”

### F.7.17.1 What is a syntax-example of a Select-total aggreg?

An illustrative example of a fairly complete **Select-total\_aggreg**<sup>39</sup> is:

```
SELECT   Len(Max(Capital & "_Cap")) AS Out_fld
        , 2 * Len(Out_fld)          AS Nonsense_2
        , 5 * Avg(5+Temp_Min)      AS Nonsense_5
FROM T_Capital_Temps
WHERE Capital <> "Beijing"
HAVING Min(3*Temp_Min) < 0
```

There are **three** “**SELECT**” expressions:

- The **three** “**SELECT**” expressions are assigned an **output field name** using the optional “**AS**” clause. The **three output field names** are “**Out\_fld**”, “**Nonsense\_2**” and “**Nonsense\_5**”.
- The “**SELECT**” expression for **output field** “**Out\_fld**” uses as an **element** the SQL aggregate function “**Max()**” over an **expression** that uses as an **element** the **input field name** “**Capital**”.
- The “**SELECT**” expression for **output field** “**Nonsense\_2**” uses as an **element** the **output field name** “**Out\_fld**”.
- The “**SELECT**” expression for **output field** “**Nonsense\_5**” uses as an **element** the SQL aggregate function “**Avg()**” over an **expression** that uses as an **element** the **input field name** “**Temp\_Min**”.

The **input** record-list in the “**FROM**” clause is:

- The Table name “**T\_Capital\_temps**” (click *F.10.5*).

The **elements** of the “**WHERE**” *Boolean-expression* are:

- The **input field name** “**Capital**”.

The **elements** of the “**HAVING**” *Boolean-expression* are:

- The SQL aggregate function “**Min()**” over an **expression** that uses as an **element** the **input field name** “**Temp\_Min**”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “*F.11.2 What are the SQL color codes used in this Guide?*”.

---

<sup>39</sup> This is the Query “**F\_Select\_w\_total\_aggreg\_Syn**” from the “**Company\_Database.accdb**” file.

### F.7.17.2 What are the formal rules (syntax) to write a Select-total\_aggreg?

A correct **Select-total\_aggreg** has to be written in the following way:

```

SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , SQL_agg_func(exp_o11(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_o1y(INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
    [, ...
    , Output-exp_n( Output-field-names
                  , SQL_agg_func(exp_on1(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_onz(INOUT-field-names))
                  ) [ AS Output-field-name_n ] ]

FROM {   [(] {Table-name or Query-name} [)]
      or [ {Table-name or Query-name} [AS Input-record-list-name] ]
      or [ ( {Select-opr or Union-opr } ) [AS Input-record-list-name] ]
      or [(] Inner-or-Outer-Join-opr [)] or Cross-Join-opr [ ] }

[ WHERE Where-Boolean-exp(Input-field-names) ]

[ HAVING Having-Boolean-exp( SQL_agg_func(exp_h1(Input-field-names))
                          , ...
                          , SQL_agg_func(exp_ht(Input-field-names)) ) ]

[ORDER BY clause]

```

The words in **bold** font are SQL **keywords**. The elements enclosed in square brackets “[ ]” are optional, and you may use each of these elements or not, depending on your desired result from the **Select** operation. The elements separated with “or” are alternative options. Each list of alternative options for an element is enclosed between curly braces “{ }” when the element is **not optional** and between square brackets “[ ]” when the element is **optional**. Some curly braces “{ }” and square brackets “[ ]” are colored just to make it easier to see which ones are paired.

The elements in gray text are the ones I advise that you do not use:

- I advise you **do not use** the optional “**DISTINCT**” clause because in this case only **one** modified record is produced, and therefore, this clause does not make any sense. For this reason, I will not be explaining this clause in this section.
- I advise you **do not use** the optional “**TOP**” clause because in this case only **one** modified record is produced, and therefore, the “**TOP**” clause does not make much sense.
- I advise you **do not use** the optional “**DISTINCTROW**” clause because it is not compatible with SQL server, and because you can get its same functionality in an easy way. If you want to know more, you may click “[F.7.8 What is the “DISTINCTROW” clause of a Select?](#)”.
- I advise you **do not use** the optional “**ALL**” clause because it does not have any effect.
- I advise you **do not use** the optional “**ORDER BY**” clause because in this case only **one** modified record is produced, and therefore, this clause does not make any sense. For this reason, I will not be explaining this clause in this section.

All the terms above with a trailing “-**exp**” are **expressions**. The **terms** enclosed between the **parentheses** of each expression are the **elements** that can be used to build that specific expression, by **combining** these **elements** with functions (excluding SQL aggregate), value operators and constants.

All the terms above with a trailing “-**opr**” are SQL operations.

The “**Input-field-names**” are **all** the **input** field names from the **input** record-list.

The “**INOUT-field-names**” are the **1** to **v** “**Input-field-names**” and/or the **1** to **n** “**Output-field-names**”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

I will now explain how to write the different clauses of the **Select-total\_aggreg** in the order they appear in the SQL code, because I think this is more convenient for syntax purposes.

### How do I write the “**SELECT**” clause?

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , SQL_agg_func(exp_o11(INOUT-field-names))
                  , ...
                  , SQL_agg_func(exp_o1y(INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
[ , ...
  , Output-exp_n( Output-field-names
                , SQL_agg_func(exp_on1(INOUT-field-names))
                , ...
                , SQL_agg_func(exp_onz(INOUT-field-names))
                ) [ AS Output-field-name_n ] ]
```

After the “**SELECT**” keyword (and after the optional clauses “**DISTINCT**”, “**TOP**” and/or “**DISTINCTROW**”, in case you are using them) you write the “**SELECT**” **expressions** “**Output-exp()**” separated with commas “**,**”. Each “**SELECT**” **expression** that you write will produce **one output** field.

Each “**Output-exp()**” is built over the **elements**:

- “**Output-field-names**” **1** to **n**  
This is, **other output field names** (as long as you **do not** create a **circular** reference, click [F.7.14](#)).
- “**SQL\_agg\_func(exp(INOUT-field-names))**”  
This is, any number of SQL aggregate functions (click [F.7.18](#)) each having as argument its specific **expression** over the “**Input-field-names**” and/or over the “**Output-field-names**” (as long as you **do not** create a **circular** reference, **nor** a **nested** SQL aggregate function, click [F.7.14](#)).

Although you can use “**Output-field-names**” in the “**SELECT**” **expression**, remind that you **cannot** use them in the “**HAVING**” *Boolean*-expression.

If you add after a “**SELECT**” **expression** the keyword “**AS**” followed by a **name**, this will be the **name** of this **output** field. If you do **not add** this “**AS**” clause, then you have two cases:

If the “**Output-exp\_i()**” consists **exactly** of a single **input** field **name**, then the

**output** field **name** will be that **input** field **name**.

Otherwise, MS-Access will assign to that “**Output-exp<sub>i</sub> ()**” the **output** field **name** “**ExprXXXX**”, where “**XXXX**” is a four-digit integer number.

Be aware that an expression “**exp<sub>o</sub> ()**” that is the argument of a given SQL aggregate function **cannot** contain any SQL aggregate function. This **cannot** happen in a direct manner, **nor through a reference** to another “**Output-field-name**”.

If you want to know more about the “**SELECT**” clause, you may click “[F.7.5 What are the \*output fields\* \(“\*SELECT\*” clause\) of a \*Select\*?](#)”.

### How do I write the “**TOP**” and “**DISTINCTROW**” clauses?

Notice that using “**TOP**” does not make much sense because a **Select-total<sub>agg</sub>** produces **zero** or **one** record.

The optional “**TOP**” clause is written by adding the “**TOP int**” after the “**SELECT**” keyword. If you use the **percent** variant, then it is written by adding “**TOP int PERCENT**” after the “**SELECT**” keyword. In case “**DISTINCT**” is used, then the “**TOP**” clause is written after the “**DISTINCT**” keyword. The symbol “**int**” stands for a positive integer constant, not greater than 100 for the **percent** variant.

If you want to know more about the “**TOP**” clause, you may click “[F.7.13 What is the “\*TOP\*” clause of a \*Select\*?](#)”.

I advise you do not use the optional “**DISTINCTROW**” clause. If in spite of my advice you want to use it, you just write the keyword “**DISTINCTROW**” after the “**SELECT**” keyword (or after the “**TOP**” clause, in case it is used). Remind that you cannot use **both** “**DISTINCT**” and “**DISTINCTROW**” in the same **Select** operation.

If you want to know more about the “**DISTINCTROW**” clause, you may click “[F.7.8 What is the “\*DISTINCTROW\*” clause of a \*Select\*?](#)”.

### How do I write the “**FROM**” clause?

```
FROM {      [ ( [ {Table-name or Query-name} ] ) ]
         or [ {Table-name or Query-name} ] [AS Input-record-list-name] ]
         or [ ( {Select-opr or Union-opr } ) [AS Input-record-list-name] ]
         or [ [ ( Inner-or-Outer-Join-opr ) ] or Cross-Join-opr ] }
```

Right after the “**FROM**” keyword you write the **input record-list**. The **input** record-list can be a Table name or a Query name or a **Join** operation or a **Select** operation or a **Union** operation. Notice this is **different** from the writing rules (syntax) of the **Join** operation.

Let me clarify the rules for **parentheses** and the “**AS**” clause, depending on what is the **input record-list** after the “**FROM**” keyword:

- **A Table name or Query name**  
It **may** be enclosed between parentheses, or, it **may** have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one of the **Join** operation.

- **A Select operation or Union operation**  
It **must** be enclosed between parentheses. It **may also** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the one of the **Join** and **Union** operations.
- **A Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.
- **A Cross-Join operation**  
It **must not** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of the **Join** operation.

If you want to know more about the “**FROM**” clause, you may click “[F.7.4 What is the input record-list \(“FROM” clause\) of a Select?](#)”.

### How do I write the “WHERE” clause?

```
WHERE Where-Boolean-exp (Input-field-names)
```

The “**Where-Boolean-exp ()**” expression that you write after the “**WHERE**” keyword is built by combining the **input field names** with functions (**excluding** SQL aggregate), value operators and constants.

If you want to know more about the “**WHERE**” clause, you may click “[F.7.7 What is the “WHERE” clause of a Select?](#)”.

### How do I write the “HAVING” clause?

```
HAVING Having-Boolean-exp ( SQL_agg_func (exp_h1 (Input-field-names) )
                          , ...
                          , SQL_agg_func (exp_ht (Input-field-names) ) )
```

The “**Having-Boolean-exp ()**” expression in the “**HAVING**” clause is built over “**SQL\_agg\_func (exp (Input-field-names) )**”. This is, any number of **SQL aggregate functions** (click [F.7.18](#)) each having as argument its specific **expression** over the “**Input-field-names**”.

If you want to know more about the “**HAVING**” clause, you may click “[F.7.10 What is the “HAVING” clause of Select-group by aggreg or Select-total aggreg?](#)”.

### Final remarks

The outermost **Select-total\_aggreg** operation of a Query may be enclosed between parentheses.

Finally, if you want to see the **dataflow** diagram of the **three types** of **Select** operations, you may click “[F.7.3 What is the dataflow of a Select?](#)”.

## F.7.18 What is an SQL aggregate function?

An **SQL aggregate function** returns **one result** from a **group** of **values**, where **each value** in the **group** is computed by applying the **expression** in the argument of the SQL aggregate function to **one input record** from a **group** of **input records**. Therefore, an SQL aggregate function returns **one result** from a **group** of **input records**. The argument of an SQL aggregate function is an **expression** over the “**Input-field-names**” (either directly, or **indirectly** though the usage of “**Output-field-names**”).



SQL aggregate functions can **only** be used in some:

- “**SELECT**” **expressions**
- “**TRANSFORM**” **expression**
- “**HAVING**” *Boolean* expression
- “**ORDER BY**” expressions

within a **Select-group\_by\_aggreg** or **Select-total\_aggreg** operations (click *F.7.2*), or within a **Transform** operation (click *F.10*).

SQL aggregate functions **cannot** be **directly** used in your VBA code (although they can be part of SQL code that is invoked from VBA code).

Notice that **SQL aggregate functions** have the **same basic functionality** as **domain aggregate** functions (click *G.6.2*). However, they have **relevant differences**. One such difference is that **SQL aggregate functions** can **only** be used in **some** expressions **belonging** to a few SQL clauses. Conversely, **domain aggregate** functions can be **directly** used in your user-defined VBA functions and subroutines, in your macros, in **any** expression within your SQL code and in your calculated controls.

MS-Access allows you to use the following built-in SQL aggregate functions:

- **Count(\*)**  
Click “*F.7.18.1 What is the “Count(\*)” SQL aggregate function?*”.
- **Count(expression(field-names))**  
Click “*F.7.18.2 What is the “Count()” SQL aggregate function?*”.
- **First, Last(expression(field-names))**  
Click “*F.7.18.3 What are the “First()” and “Last()” SQL aggregate functions?*”
- **Min, Max(expression(field-names))**  
Click “*F.7.18.4 What are the “Min()” and “Max()” SQL aggregate functions?*”.
- **Sum, Avg(expression(field-names))**  
Click “*F.7.18.5 What are the “Sum()” and “Avg()” SQL aggregate functions?*”.
- **StDev, StDevP, Var, VarP(expression(field-names))**  
Click “*F.7.18.6 What are the “StDev()”, “StDevP()”, “Var()” and “VarP()” SQL aggregate functions?*”.

MS-Access **does not** allow to write user-defined SQL aggregate functions, so you can only use the above built-in SQL aggregate functions that are provided by MS-Access. However, there are some **useful tricks** to get the **same result** as if the SQL aggregate functions “**AllNull()**”, “**AllEqual()**”, “**Or()**” and “**And()**” would exist. If you want to know more about this, you may click “*K.6.14 What are useful tricks with SQL aggregate functions?*”.

An **exception-value** (click *J.15*) in one, or more, record fields being processed by **any aggregate function** will **crash** the Query. The exception are the “**Count(\*)**” and “**DCount(“\*”)**” aggregate functions that count the **records** in each **group** of records, regardless of their field values, and will return a correct value even if one or more record fields contain exception-values.



### F.7.18.1 What is the “Count (\*)” SQL aggregate function?

**Count (\*)** returns the **number of records** in each **group** of records. Notice that **all the records** in each **group** of records are counted, even records with one, several or **all Null** field values.

**Count (\*)** always returns a **non-Null integer-like** value. If a **group** of records contains no records, then **Count (\*)** returns the value “0” for that **group** of records.

### F.7.18.2 What is the “Count ()” SQL aggregate function?

**Count (expression (field-names) )** returns the **number of records** in each **group** of records that produce a **non-Null result** in the argument “expression ()”. Notice that records whose result in “expression ()” is **Null are not counted**.

**Count ()** always returns a **non-Null integer-like** value. If a **group** of records contains no records, then **Count ()** returns the value “0” for that **group** of records.

**Count ()** can be used over **any Table field type and any VBA data type**, including **AutoNumber, Long Text, OLE Object, Hyperlink and Attachment**.

### F.7.18.3 What are the “First ()” and “Last ()” SQL aggregate functions?

**First (expression (field-names) )** and  
**Last (expression (field-names) )**

return the result of the argument “expression ()” calculated over the fields of the **first** or **last** (respectively) **record** from each **group** of records.

**First ()** and **Last ()** can be used over **any Table field type and any VBA data type**, including **AutoNumber, Long Text, OLE Object, Hyperlink and Attachment**.

**First ()** and **Last ()** return the same data type returned by its argument “expression ()”.

**First ()** and **Last ()** return **Null** for a **group** of records in case its argument “expression ()” returns **Null** over the **first or last** record (**respectively**) in that **group** of records. **First ()** and **Last ()** also return **Null** for a **group** of records in case the **group** of records contains **no records**.

Notice that **what is the first or last record** depends on the **internal record-list ordering** of each **group**, as managed by MS-Access. Therefore, the first or last record **does not** depend on the “**ORDER BY**” clause that is used in the **input record-list** of the **Select** or **Transform** operations. Therefore, “**First ()**” and “**Last ()**” return one **arbitrary** record from each **group** of records.

### F.7.18.4 What are the “Min ()” and “Max ()” SQL aggregate functions?

**Min (expression (field-names) )** and  
**Max (expression (field-names) )**

return the **minimum** or **maximum** (respectively) value among the **non-Null** results of its argument “expression ()” calculated over the fields of each record from its **group** of records.

**Min ()** and **Max ()** can be used over any **string** data/field type and any **numeric-like** data/field type (click *F.7.18.7*).

Therefore, **Min()** and **Max()** can be used over the VBA data types *Byte*, *Integer*, *Long*, *LongLong*, *Currency*, *Single* and *Double*, plus their equivalent Table field types-sizes **Number-Byte**, **Number-Integer**, **Number-Long Integer**, **Large Number**, **Currency**, **Number-Single** and **Number-Double**.

**Min()** and **Max()** cannot be used over **Long Text**, **Hyperlink**, **OLE Object** and **Attachment**.

The value **True/Yes/On** or **ticked** is considered as number “-1” and **False/No/Off** or **unticked** as number “0”, while **datetime** values are interpreted as the corresponding number values (click *D.4.5*).

**Min()** and **Max()** return the same data type returned by its argument “expression”.

**Min()** and **Max()** return **Null** for a **group of records** in case its argument “expression” returns **Null** over **each and every** record in that **group of records**. **Min()** and **Max()** also return **Null** for a **group of records** in case the **group of records** contains **no records**.

If **Min()** or **Max()** are used over any of its unsupported data types, the Query will **crash**, and MS-Access will show the error message:

*“Cannot have Memo, OLE, or Hyperlink Object fields in aggregate argument (Arg\_expr).”*

where “*Arg\_expr*” is the expression in the argument of the aggregate function that caused the error. Remind that “**Memo**” is the name used for “**Long Text**” in older MS-Access versions.

Notice that the minimum or maximum value is calculated based on the specific data/field type of the expression. For example: “**50**” is **larger** than “**2000**” if they are text strings, but “**50**” is obviously **smaller** than “**2000**” if they are numbers. You must therefore be sure of what is the data type of its argument “expression”, or you will not get the results that you expect.

### **F.7.18.5 What are the “Sum()” and “Avg()” SQL aggregate functions?**

**Sum**(expression(field-names)) and  
**Avg**(expression(field-names))

return the **sum** or the **average** (respectively) of all the **non-Null** results of its argument “expression” calculated over the fields of each record from its **group of records**. Notice that records that produce a **Null** result in its argument “expression” are **ignored** in the calculation of the **sum** or **average** (respectively).

**Sum()** and **Avg()** can be used over any **numeric-like** data/field type (click *F.7.18.7*).

Therefore, **Sum()** and **Avg()** can be used over the VBA data types *Boolean*, *Byte*, *Integer*, *Long*, *LongLong*, *Currency*, *Single*, *Double* and *Date*, plus their equivalent field types-sizes: **Yes/No**, **Number-Byte**, **Number-Integer**, **Number-Long Integer**, **Large Number**, **Currency**, **Number-Single**, **Number-Double** and **Date/Time**

**Sum()** and **Avg()** cannot be used over *String*, **Short Text**, **Long Text**, **Hyperlink**, **OLE Object** and **Attachment**.

The value **True/Yes/On** or **ticked** is considered as number “-1” and **False/No/Off** or **unticked** as number “0”, while **datetime** values are interpreted as the corresponding number values (click *D.4.5*).

**Sum ()** and **Avg ()** return the same data type returned by its argument **its argument** “*expression ()*”.

**Sum ()** and **Avg ()** return **Null** for a **group of records** in case **its argument** “*expression ()*” returns **Null** over **each and every** record in that **group of records**.

**Sum ()** and **Avg ()** also return **Null** for a **group of records** in case the **group of records** contains **no records**.

If **Sum ()** or **Avg ()** is used over a **String** or **Short Text** value, the Query will **crash**, and MS-Access will show the error message:

*“Data type mismatch in criteria expression.”*

If **Sum ()** or **Avg ()** is used over any other of its other unsupported data types, the Query will **crash**, and MS-Access will show the error message:

*“Cannot have Memo, OLE, or Hyperlink Object fields in aggregate argument (Value\_expr).”*

where “*Value\_expr*” is the expression in the argument of the **Sum ()** or **Avg ()** function that caused the error. Remind that “**Memo**” is the name used for “**Long Text**” in older MS-Access versions.

#### **F.7.18.6 What are the “StDev ()”, “StDevP ()”, “Var ()” and “VarP ()” SQL aggregate functions?**

**StDev (expression (field-names) )**,  
**StDevP (expression (field-names) )**,  
**Var (expression (field-names) )** and  
**VarP (expression (field-names) )**

return, respectively, the **sample standard deviation**, the **population standard deviation**, the **sample variance**, and the **population variance**, over all the **non-Null** results of its argument “*expression ()*” calculated over the fields of each record in its **group** of records. Notice that records that produce a **Null** result in its argument “*expression ()*” are **ignored** in the computations of these functions.

**StDev ()**, **StDevP ()**, **Var ()** and **VarP ()** are collectively called “**Two-value**” SQL aggregate functions, because they require **at least two non-Null** results from its **group** of records to return a **non-Null** result.

**StDev ()**, **StDevP ()**, **Var ()** and **VarP ()** can be used over any numeric-like Table field type or any numeric-like VBA data type (click *F.7.18.7*). Therefore, they cannot be used over **String**, **Short Text**, **Long Text**, **Hyperlink**, **OLE Object** and **Attachment**. The value **True/Yes/On** or **ticked** is considered as number “-1” and **False/No/Off** or **unticked** as number “0”, while **Date/Time** or **Date** values are interpreted as the corresponding number values (click *D.4.5*).

**StDev ()**, **StDevP ()**, **Var ()** and **VarP ()** return the same data type returned by its argument “*expression ()*”.

**StDev()**, **StDevP()**, **Var()** and **VarP()** return **Null** for a **group of records** in case its argument “*expression()*” returns **Null** over **all**, or **all but one**, records in that **group of records**. **StDev()**, **StDevP()**, **Var()** and **VarP()** also return **Null** for a **group of records** in case the **group of records** contains **zero** or **one** records.

If **StDev()**, **StDevP()**, **Var()** or **VarP()** is used over a **String** or **Short Text** value, the Query will **crash**, and MS-Access will show the error message:

*“Data type mismatch in criteria expression.”*

If **StDev()**, **StDevP()**, **Var()** or **VarP()** is used over any of its other unsupported data types, the Query will **crash**, and MS-Access will show the error message:

*“Cannot have Memo, OLE, or Hyperlink Object fields in aggregate argument (Value\_expr).”*

where “*Value\_expr*” is the expression in the argument of the SQL aggregate function that caused the error. Remind that “**Memo**” is the name used for “**Long Text**” in older MS-Access versions.

### **F.7.18.7 What is a summary and grouping of aggregate functions?**

SQL aggregate functions (and their **dual** domain aggregate functions, click *G.6.2*) are usually classified as follows:

- **Two-value** aggregate functions  
**Two value** aggregate functions are **StDev()**, **DStdev**, **StDevP()**, **DStDevP()**, **Var()**, **DVar()**, **VarP()** and **DVarP()**.  
**Two-value** aggregate functions require **at least two non-Null input** values to return a **non-Null** value.  
**Two-value** aggregate functions require that their argument is of a **numeric-like** data/field type and return a **numeric-like** data/field type.
- **Numeric-like** aggregate functions  
**Numeric-like** aggregate functions are the **two-value** ones plus **Avg()**, **DAvg()**, **Sum()** and **DSum()**.  
**Numeric-like** aggregate functions require that their argument is of a **numeric-like** data/field type and return a **numeric-like** data/field type.
- **Calculation** aggregate functions  
**Calculation** aggregate functions are **numeric-like** ones plus **Min()**, **DMin()**, **Max()** and **DMax()**.  
**Calculation** aggregate functions are the ones that perform some calculation over their **group** of records.  
**Min()**, **DMin()**, **Max()** and **DMax()** require that their argument is of **string** or **numeric-like** data/field types.  
**Calculation** aggregate functions return a **value** of the **same data/field type** (either **string** or **numeric-like**) as the expression of their argument.
- **Value** aggregate functions  
**Value** aggregate functions are **calculation** ones plus **First()**, **DFirst()**, **DLookup()**, **Last()** and **DLast()**.  
**First()**, **DFirst()**, **DLookup()**, **Last()** and **DLast()** work over **any** data/field type and return an **integer**.

- All aggregate functions  
All aggregate functions are **value** ones plus `Count()`, `DCount()`, `Count(*)` and `DCount(" * ")`.  
`Count()`, `DCount()`, `Count(*)` and `DCount(" * ")` work over **any** data/field type and return an integer.

## F.8 What is a Join operation and how do I write it?

A **Join operation** is an SQL operation performed with a **Join** operator (click *F.8.2*) plus its corresponding operands. Therefore, a **Join operation** is the complete SQL code associated to a **Join** operator.

A simple example of a **Join** operation<sup>40</sup> is:

```
T_House_owners
INNER JOIN
T_Car_owners
ON Houses.ID = Cars.ID ;
```

The above **Join operation** produces **output** records joining **every left input** record from the Table `T_House_owners` with **every right input** record from the Table `T_Car_owners`, but only when both records have the same **value** in their `ID` field. The field **values** of each **output record** are **all** the field **values** (in the same order) from its originating **left input** record **followed** by all the field **values** (in the same order) from its originating **right input** record.

A **Join** operation **cannot** be the **outermost** operation in a Query. If you have a **series** of nested **Join operations**, the outermost **Join** operation **must** be enclosed in a **Select** operation.

If you want to know more about a **Join operation**, you may click:

- [“F.8.1 What is “joining” two ordered records?”](#)
- [“F.8.2 What are the Join operators?”](#)
- [“F.8.3 What are the input record-lists of a Join?”](#)
- [“F.8.4 What are the output fields of a Join?”](#)
- [“F.8.5 What is the output record-list of a “ , ” Cross-Join?”](#)
- [“F.8.6 What is the output record-list of an “INNER JOIN”?”](#).
- [“F.8.7 What is the output record-list of an Outer-Join \(“LEFT JOIN” or “RIGHT JOIN”\)?”](#)
- [“F.8.8 What is the output record-list of a Full-Outer-Join?”](#)
- [“F.8.10 How do I write a correct \(syntax\) Join?”](#)

### F.8.1 What is “joining” two ordered records?

“Joining” two **ordered** records, a **left “l”** record and a **right “r”** record, produces **one** larger **output record** by concatenating the **fields** and the field **values** of these two

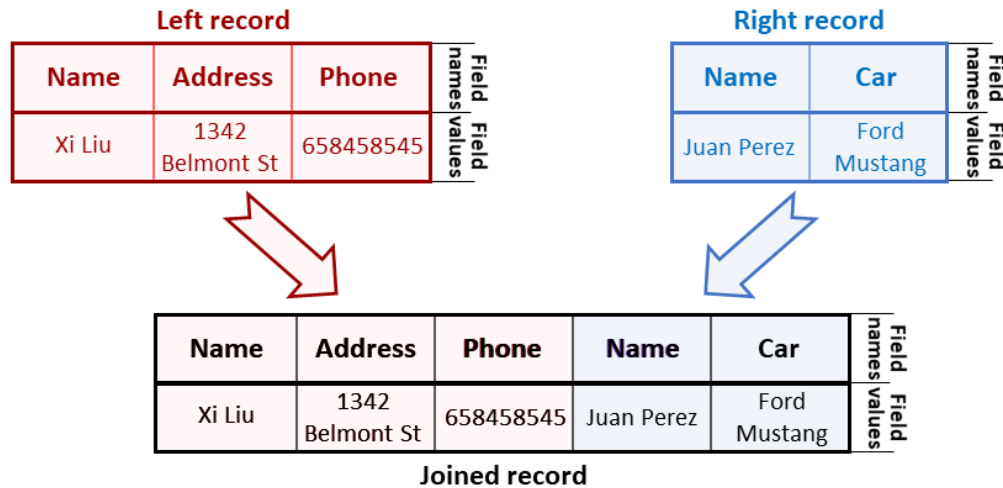
---

<sup>40</sup> A Join operation (or a series of Join operations) **must always** be enclosed in a **Select** operation. If you want to run this Join operation, you have to enclose it in a **Select** operation.

ordered records.

Joining a **left** “**l**” record and a **right** “**r**” record is also called **joining** the **ordered** record pair (**l, r**).

The following picture graphically shows the result of joining a **left** and a **right** record:



Notice that **joining** records is **not commutative**. It is not the same **joining** the record pair (**a, b**) than **joining** the record pair (**b, a**), because the **order** of **output** fields and field **values** is different. To remind you of this I will be frequently adding the term “**ordered**” when talking about joined record pairs along this chapter.

I now explain in more detail the resulting **output** record of **joining** an **ordered** pair of records.

### What are the resulting fields of joining an ordered record pair?

When you **join** an **ordered** record pair (**l, r**), the **fields** of the **joined output** record are **all** the **fields** (in the same order) from “**l**” followed by **all** the **fields** from “**r**”. Each **output** field keeps the same **properties** (field name, field type, ...) it had in the corresponding **left** or **right** record.

For example, if the **left** (first) record has **fields** “**Name**”, “**Address**” and “**Phone**”, and the **right** (second) record has **fields** “**Name**” and “**Birthdate**”, the corresponding **joined output** record will have **fields** “**Name**”, “**Address**”, “**Phone**”, “**Name**” and “**Car**”. Notice that the field name “**Name**” is **repeated**, which happens **very** frequently. This is called a “**duplicated field name**”.

In case the **output** records have **duplicated field names**, in order to avoid ambiguity, any reference to a **duplicated field name** must be **qualified**. Remind that qualifying a field name (click C.2.2) consists of prefixing the field name with the **name** of the **input** record-list that contained that field name, with an intermediate period “.” character. Notice that even if **two** fields have the **same** unqualified **field name**, they are **two different fields** and their field **properties** (e.g., field type) and field **values** are completely **independent**, and may be completely different (or not).

### What are the resulting field values of joining an ordered record pair?

When you **join** an **ordered** record pair (**l, r**), the field **values** of the **joined output** record are **all** the field **values** (in the same order) from “**l**” followed by **all** the field **values** (in the same order) from “**r**”.



For example, joining a **left** “**l**” record having field values “**Xi Liu**”, “**1342 Belmont St**” and “**658458545**” with a **right** “**r**” record having field values “**Juan Perez**” and “**Ford Mustang**”, produces a **joined output** record having field values “**Xi Liu**”, “**1342 Belmont St**”, “**658458545**”, “**Juan Perez**” and “**Ford Mustang**”.

## F.8.2 What are the Join operators?

The **Join** operators are **consulting** (click *F.6*) operators that work over **two input** record-lists, plus another possible operand that is not a record-list.

A simplified view of writing (syntax) a **Join operation** is:

```

Left-Input-record-list AS Left_name
Join-Operator
Right-Input-record-list AS Right_name
[ ON ON-Boolean-expression(Left_name.x, Right_name.y) ]

```

The “**ON**” clause enclosed between square brackets “[ ]” **cannot** be used with the “**,**” **Cross-Join** operator and it is **mandatory** for all the other **Join** operators.

The **Join** operators **join, one with one, some** records from its **left** (first) “**L**” **input** record-list with **some** records from its **right** (second) “**R**” **input** record-list. On some cases, they also join one **input** record with one **void** record.

Remind that **joining** (click *F.8.1*) a **left** “**l**” record and a **right** “**r**” record produces **one** larger **output** record that has the **fields** and field **values** of the **left** record followed (i.e., to its left) by the ones of the **right** record.

If you want to know the writing rules (syntax) of a **Join**, you may click “*F.8.10 How do I write a correct (syntax) Join?*”.

All the **Join** operators produce the same **output fields**. If you want to know more about this, you may click “*F.8.4 What are the output fields of a Join?*”.

There are four **Join operations**, and four **Join operators** (but there is no one-to-one correspondence):

- **Cross-Join operation**: operator “**,**” (i.e., just one comma character)  
Joins **all** (**l**, **r**) pairs.
- **Inner-Join operation**: operator “**INNER JOIN**”  
Joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs.
- **Outer-Join operation**: operators “**LEFT JOIN**” and “**RIGHT JOIN**”  
A “**LEFT JOIN**” joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, **plus**, all the (**l<sub>nm</sub>**, **void**) **non-matching** pairs. A “**RIGHT JOIN**” joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, **plus**, all the (**void**, **r<sub>nm</sub>**) **non-matching** pairs.
- **Full-Outer-Join operation**: does not have an operator in MS-Access  
A **Full-Outer-Join** joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, **plus**, all the (**l<sub>nm</sub>**, **void**) **non-matching** pairs, **plus**, all the (**void**, **r<sub>nm</sub>**) **non-matching** pairs.

A **void** record has **all** its fields **void**. A **void** field has **Null** pointers to **all** the **Table fields** in its expanded “**SELECT**” **expression**. An expanded “**SELECT**” **expression** is the one where **all** the **field names** from the enclosed SQL operations or auxiliary Queries are **replaced, recursively**, by the corresponding “**SELECT**” **expression**. Therefore, an expanded “**SELECT**” **expression** is composed of **Table field names** combined with



functions, operators and constants, and it **does not** contain any field name from an SQL expression or auxiliary Query.

One main difference between the four **Join operations** is **which** records are **joined** to produce the **output** record-list. The other difference is that the **Inner-Join**, **Outer-Join** and **Full-Outer-Join operations** require an “ON” clause with a *Boolean* expression, while the **Cross-Join** does not have the “ON” clause. Let me now explain the result of the **four Join operations** and the **four Join operators** over a **left input** record-list “L” and a **right input** record-list “R”. In the explanation below “l” is a record from “L” and “r” is a record from “R”.

- **Cross-Join operation:** operator “,” (i.e., just one comma character)

A **Cross-Join** joins **all** (l, r) pairs.

In more detail, a **Cross-Join** joins **all** ordered record pairs (l, r) in the **set of record pairs** (L, R). This is called the “Cartesian product”.

A **Cross-Join** does not cause the pointers to the **Table fields** in its “SELECT” **expressions to become Null**, and therefore it **does not create Nulls** (but Nulls existing in the **input** records **will stay** in the corresponding **output** records).

Notice that the SQL keyword for the **Cross-Join** operator is just a comma (“,”).

The **Cross-Join** operator is also called the **Full-Join** operator.

If you want to know more about the **output** record-list of a **Cross-Join**, you may click “F.8.5 *What is the output record-list of a “,” Cross-Join?*”.

- **Inner-Join operation:** operator “INNER JOIN”

An “INNER JOIN” joins **all** (l<sub>m</sub>, r<sub>m</sub>) **matching** pairs.

We say that a (l<sub>m</sub>, r<sub>m</sub>) pair **matches** when the **field values** of “l<sub>m</sub>” and “r<sub>m</sub>” **jointly** return *True* in *Boolean*-expression of the “ON” clause of the “INNER JOIN” operator. If the result is other than *True*, then (l<sub>nm</sub>, r<sub>nm</sub>) is a **non-matching** pair.

In more detail, a “INNER JOIN” joins **all** ordered record pairs (l<sub>m</sub>, r<sub>m</sub>) in **each** and **every** of the “N” **sets of matching ordered record pairs** (L<sub>mi</sub>, R<sub>mi</sub>).

In this statement, **each** “L<sub>mi</sub>” is a **subset** of “L” and **each** “R<sub>mi</sub>” is a **subset** of “R”, such that **all** records in “L<sub>mi</sub>” **match** with **all** records in “R<sub>mi</sub>”. The **index** “i” goes from 1 to “N”. It is also required that the “N” (L<sub>mi</sub>, R<sub>mi</sub>) **pairs of record sets** contain the same elements as all the **matching record pairs** (l<sub>m</sub>, r<sub>m</sub>) between “L” and “R”.

This second definition of “INNER JOIN” may seem a little strange now, but it will make sense when you see the picture at the end of this section comparing the results from the different **Join** operators and also when explaining the number of **output** records (click F.8.6.3) from an “INNER JOIN”.

An “INNER JOIN” **does not** cause the pointers to the **Table fields** in its “SELECT” **expressions to become Null**, and therefore it **does not create Nulls** (but Nulls existing in the **input** records **will stay** in the corresponding **output** records).

Notice that an “INNER JOIN” produces a **subset** of the **output** records produced by a **Cross-Join**. The **subset** corresponds to those **Cross-Join’s output** records arising from **matching input** records.

If you want to know more about the **output** record-list of an “**INNER JOIN**”, you may click “*F.8.6 What is the output record-list of an “INNER JOIN”?*”.

- **Outer-Join operation:** operators “**LEFT JOIN**” and “**RIGHT JOIN**”

A “**LEFT JOIN**” joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, **plus**, **all** the (**l<sub>nm</sub>**, **void**) **non-matching** pairs.

We say that that (**l<sub>nm</sub>**, **void**) is a **non-matching** pair when **all** pairs in the set (**l<sub>nm</sub>**, **R**) **do not match**. This is, when the “**l<sub>nm</sub>**” record does not match with **any** record in “**R**”.

In more detail, a “**LEFT JOIN**” joins the same (**l<sub>m</sub>**, **r<sub>m</sub>**) ordered record pairs as an “**INNER JOIN**”, **plus**, all the ordered record pairs (**l<sub>nm</sub>**, **void**) in the set (**L<sub>nm</sub>**, **void**).

In this statement, the record-list “**L<sub>nm</sub>**” contains **all** the **non-matching** “**l<sub>nm</sub>**” **input** records. This is, “**L<sub>nm</sub>**” is the **subset** of “**L**” such that **all** the pairs (**l<sub>nm</sub>**, **R**) **do not match**.

A **void** record has **all** its fields **void**. A **void** field has **Null** pointers to **all** the **Table fields** in its **expanded “SELECT” expression**.

A “**RIGHT JOIN**” is exactly the same as a “**LEFT JOIN**”, with the only difference that the **input** record-list that produces the additional set of records, is the **right** (**second**) one instead of the **left** (**first**) one. Therefore, it makes **Null** **all** the pointers to the **Table fields** in the “**SELECT**” **expressions** of the **left** (**first**) “**L**” **input** record-list. This may cause a “**SELECT**” **expression** to **crash**, or to return an exception-value or a **Null**.

Therefore, a “**RIGHT JOIN**” joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) pairs that **match**, **plus**, all the (**void**, **r<sub>nm</sub>**) pairs, such that **all** pairs in the set (**L**, **r<sub>nm</sub>**) **do not match**.

Using “**LEFT JOIN**” is usually preferable to using “**RIGHT JOIN**” because it is considered that the resulting SQL code is more readable. However, a “**RIGHT JOIN**” may still be useful on some cases.

If you want to know more about the **output** record-list of the **Outer-Join** operators, you may click “*F.8.7 What is the output record-list of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?*”.

- **Full-Outer-Join operation:** does not have an operator in MS-Access

A **Full-Outer-Join** joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, **plus**, **all** the (**l<sub>nm</sub>**, **void**) and **all** the (**void**, **r<sub>nm</sub>**) **non-matching** pairs.

We say that that (**l<sub>nm</sub>**, **void**) is a **non-matching** pair when **all** pairs in the set (**l<sub>nm</sub>**, **R**) **do not match**. This is, when the “**l<sub>nm</sub>**” record does not match with **any** record in “**R**”.

We say that that (**void**, **r<sub>nm</sub>**) is a **non-matching** pair when **all** pairs in the set (**L**, **r<sub>nm</sub>**) **do not match**. This is, when the “**r<sub>nm</sub>**” record does not match with **any** record in “**L**”.

In other words, a **Full-Outer-Join** joins the same pairs as an “**INNER JOIN**”, **plus**, all the **additional** (**l<sub>nm</sub>**, **void**) pairs of a “**LEFT JOIN**”, **plus**, all the **additional** (**void**, **r<sub>nm</sub>**) pairs of a “**RIGHT JOIN**”.

A **void** record has **all** its fields **void**. A **void** field has **Null** pointers to **all** the **Table**

**fields** in its expanded “**SELECT**” expression.

MS-Access **does not provide** an operator to produce a **Full-Outer-Join**. However, I am explaining **Full-Outer-Join** because it is an important concept, and on some cases you may need to write an SQL piece of code for a **Full-Outer-Join**. If you want to know more about the **output** record-list of a **Full-Outer-Join**, you may click “*F.8.8 What is the output record-list of a Full-Outer-Join?*”.

**Full-Outer-Join** is considered undesirable because it creates **Null** pointers to **Table fields** (that need to be handled properly) and **weakens indexing** (which results in worse performance). Avoid **Full-Outer-Join** as much as possible. In many cases, what you want is to “**merge**” two record-lists into a single one, and you can get this using “**UNION ALL**” and “**GROUP BY**” instead of using a **Full-Outer-Join**. If you want to know how to “merge” record-lists using “**UNION**”, you may click “*K.6.9 How do I “merge” two record-lists?*”. If you cannot avoid writing a **Full-Outer-Join**, I explain how to do it in “*K.6.8 How do I write a Full-Outer-Join?*”.

If you want to know more about the **Join** operations and operators, you may click:

- “*F.8.2.1 How do the four Join operations compare?*How do the four Join operations compare?”
- “*F.8.2.2 Are Join operations commutative?*”

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “*F.11.2 What are the SQL color codes used in this Guide?*”.

### F.8.2.1 How do the four Join operations compare?

The summary of the four **Join operations** and the **four Join operators** is:

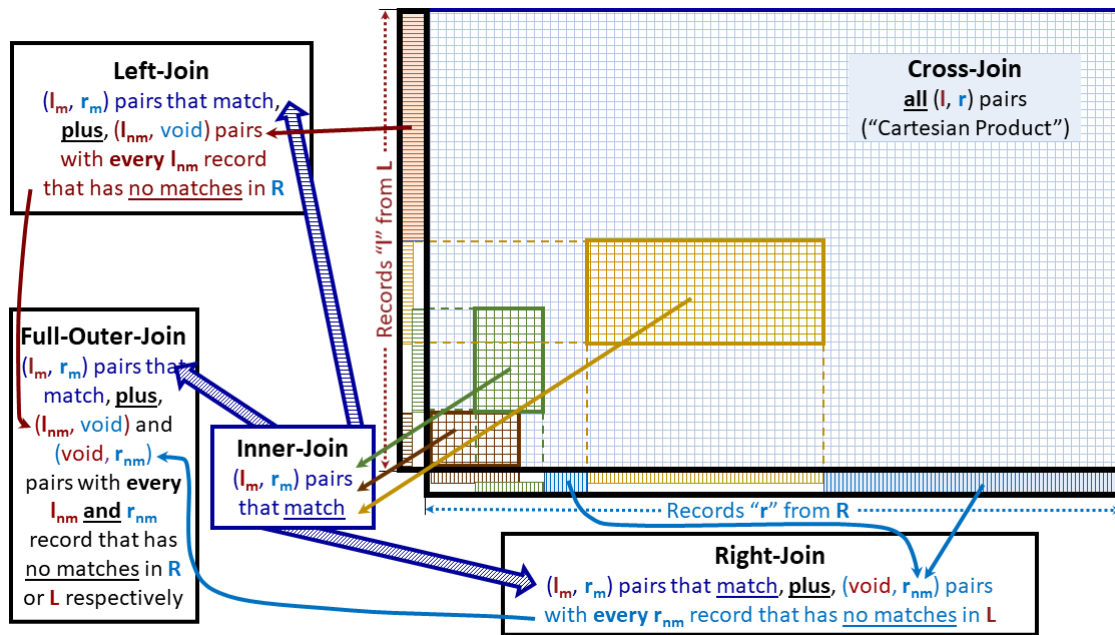
- **Cross-Join operation:** operator “,” (i.e., just one comma character)  
Joins all (**l**, **r**) pairs.
- **Inner-Join operation:** operator “**INNER JOIN**”  
Joins all (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs.
- **Outer-Join operation:** operators “**LEFT JOIN**” and “**RIGHT JOIN**”  
A “**LEFT JOIN**” joins all (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, plus, all the (**l<sub>nm</sub>**, **void**) **non-matching** pairs. A “**RIGHT JOIN**” joins all (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, plus, all the (**void**, **r<sub>nm</sub>**) **non-matching** pairs.
- **Full-Outer-Join operation:** does not have an operator in MS-Access  
A **Full-Outer-Join** joins all (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, plus, all the (**l<sub>nm</sub>**, **void**) **non-matching** pairs, plus, all the (**void**, **r<sub>nm</sub>**) **non-matching** pairs.

To better understand the results of the different **Join operations** and **Join operators**, the following picture shows the joined record pairs leading to the **output** record-list of:

**L** Join-op **R**

depending on which **Join operation** or **Join operator** you use as “**Join-op**”. Notice

that “L” and “R” are the **left** and **right input** record-lists, respectively.



The following table compares the **Join operations** and **Join operators** in terms of their having, or not, an “ON” clause with a *Boolean* expression and their possibly creating, or not, **Nulls**.

		Requires an “ON” <i>Boolean</i> -expression?	
		No	Yes
May create Nulls?	No	Cross-Join “,”	“INNER JOIN”
	Yes		Full-Outer-Join and Outer-Joins (“LEFT JOIN” and “RIGHT JOIN”)

### F.8.2.2 Are Join operations commutative?

Strictly speaking, **none** of the **Join operations** is **commutative**, this is, you **cannot** exchange their **input** record-lists and be sure that the result will be the same. The reason is that **joining** records is **not commutative** (click *F.8.1*) because if you reverse the joined records, the **order** of the **output fields** and field **values** is different.

**However**, the outermost **Join operation** must always be enclosed in a **Select operation**, and a **Select operation** works the same (except the non-advisable “**SELECT \***”) **irrespective** of the **field order** of its **input** record-list.

Therefore, the **Join operations** “,” **Cross-Join**, “**INNER JOIN**” and **Full-Outer-Join** are **commutative** (except if they are enclosed in a “**SELECT \***”). You may then **exchange** its **left** and **right input** record-lists, and you will get **exactly the same result**.

If for some reason (e.g., improving the readability of the code) you exchange the **left** and **right input** record-lists of an “**INNER JOIN**” or of a **Full-Outer-Join** be aware

that the “ON” *Boolean*-expression **may**, or **may not**, be commutative. If it is **not** commutative, you should **not exchange** the corresponding **input** field names, and rather leave the “ON” expression **unmodified**.

Let me show you the difference with an example over two Tables that I will be **also** using in other sections of this chapter *F.8*. One Table is a list of house owners and the other Table is a list of car owners. The field “ID” represents a **unique** identification number to avoid mistaking two different persons that have the same name. The example Tables are:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

If you write and run the following “INNER JOIN”<sup>41</sup>:

```
SELECT Houses.ID, Houses.Name, Cars.ID, Cars.Name
FROM
  T_House_Owners AS Houses
INNER JOIN
  T_Car_Owners AS Cars
ON Houses.ID > Cars.ID ;
```

the corresponding **output** record-list from this Query is:

F_Join_Commut_1			
Houses.ID	Houses.Name	Cars.ID	Cars.Name
6	Xi Liu	3	Peter Sellers
14	Xi Liu	3	Peter Sellers
14	Xi Liu	6	Xi Liu

If you now **exchange the two input** record-lists you get<sup>42</sup>:

```
SELECT Houses.ID, Houses.Name, Cars.ID, Cars.Name
FROM
  T_Car_Owners AS Cars
INNER JOIN
  T_House_Owners AS Houses
ON Houses.ID > Cars.ID ;
```

Notice I have kept the color of both **input** record-lists (so the **left** one is now **blue**), so you can compare more easily both SQL operations.

This second which produces **exactly the same output** record-list.

Notice that **I did not exchange** the field names “Cars.ID” and “Houses.ID” in the “SELECT” clause, nor in the “ON” expression. Notice also that this “ON” expression is **not commutative**. In case I had **mistakenly** exchanged the field names in the “ON”

<sup>41</sup> This is the Query “F\_Join\_Commut\_1” from file “Company\_Database.accdb”.

<sup>42</sup> This is the Query “F\_Join\_Commut\_2” from file “Company\_Database.accdb”.

expression, I would get the following Query<sup>43</sup>:

```
SELECT Houses.ID, Houses.Name, Cars.ID, Cars.Name
FROM
    T_Car_Owners AS Cars
INNER JOIN
    T_House_Owners AS Houses
ON Cars.ID > Houses.ID ;
```

which produces a **very different output** record-list.

### F.8.3 What are the input record-lists of a Join?

The **two input** record-lists of the **Join operators** are written at **both** sides of the **Join** operator. The **left (first) input** record-list is written just before the **Join** operator, and the **right (second) input** record-list is written just after it, as follows:

```
{ [ ( ( {Table-name or Query-name} [ ] ) ] }
or [ {Table-name or Query-name} [AS Left-name] ]
or [ ( {Select-opr or Union-opr} ) AS Left-name ]
or [ [ ( Inner-or-Outer-Join-opr [ ] ) or Cross-Join-opr ] }
{ , or INNER JOIN or LEFT [OUTER] JOIN or RIGHT [OUTER] JOIN }
{ [ ( ( {Table-name or Query-name} [ ] ) ] }
or [ {Table-name or Query-name} [AS Right-name] ]
or [ ( {Select-opr or Union-opr} ) AS Right-name ]
or [ [ ( Inner-or-Outer-Join-opr [ ] ) or Cross-Join-opr ] }
```

Both the **left** and **right input** record-lists can be written as **either** a **Table name**, a **Query name**, a **Select operation** or a **Union operation**. The **left** and **right input** record-lists **may be the same record-list**.

Notice that I have shaded above the **Join** operation with gray text. Although you can actually use a **Join** operation as an **input** record-list of the **Join operators**, the code readability becomes **poor**, and it is **very easy** to make a **mistake** with the **parentheses**. For these reasons, I advise you **avoid nesting Join operations**. If you want to do a **Join** of a **Join**, just enclose the innermost join in a **Select operation**. As an exception to this advice, you can do a **Join** of a **Join** if both operators are a **Cross-Join**, because the **Cross-Join** operator is associative, and each **Cross-Join** operation is not enclosed between parentheses, so there is no risk of making a mistake.

The rules on **parentheses** and “**AS**” clause depend on **what is the input record-list** after the “**FROM**” clause, as follows:

- **A Table name or Query name**  
It **may** be enclosed between parentheses, or, it may have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one in **Select** and **Transform** operations.
- **A Select operation or Union operation**  
It **must** be enclosed between parentheses, and it **must** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the one in **Select**, **Transform** and **Union** operations.
- **A Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause.

<sup>43</sup> This is the Query “**F\_Join\_Commut\_3**” from file “**Company\_Database.accdb**”.



This writing rule is the same in **Select** and **Transform** operations.

- **A Cross-Join operation**

It **must not** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same in the **Select** and **Transform** operations.

#### **F.8.4 What are the output fields of a Join?**

The **output fields** of all the **Join** operation are exactly the same.

The **output fields** are all the **fields** (in the same order) from the **left (first) input** record-list followed by all the **fields** (in the same order) from the **right (second) input** record-list.

The **number** of **output fields** of the **Join** operations is the **addition** of the **number** of **input** fields of its **left** and **right input** record-lists.

#### **What are the output field names of a Join?**

The **output field names** of all the **Join** operations are exactly the same.

The **output field names** are all the **field names** (in the same order) from the **left (first) input** record-list followed by all the **field names** (in the same order) from the **right (second) input** record-list.

In case the **left** and **right input** record-lists have one, or more, **equal field names**, MS-Access will **qualify** the corresponding **output field names** in the Query results shown in “**Datasheet View**”. Remind that qualifying a **field name** (click C.2.2) consists of prefixing the **field name** with the **name** of the **record-list** that contained that **field name**, with an intermediate period “.” character.

#### **What is the output field order of a Join?**

The **output field order** of all the **Join** operations is exactly the same.

The **output field order** is the **same field order** of fields from the **left (first) input** record-list followed by the **same field order** of the **right (second) input** record-list.

#### **What are the output data/field types of a Join?**

The **output field data/field types** of all the **Join** operations are exactly the same.

The **data/field type** of each **output field** is the **same** as the **data/field type** of its corresponding **input field**.

Therefore, the **data/field types** of its **output** fields are the ones (in the same order) of the fields of its **left (first) input** record-list followed by the ones (in the same order) of the fields of its **right (second) input** record-list.

#### **What are the output field values and/or output record-list of a Join?**

If you want to know what are the **output field values**, and/or what is the **output record-list** of a **Join**, they depend on the **type** of **Join**. You may click:

- “F.8.5 What is the output record-list of a “ , ” Cross-Join?”
- “F.8.6 What is the output record-list of an “INNER JOIN”?”
- “F.8.7 What is the output record-list of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?”



- “F.8.8 What is the output record-list of a Full-Outer-Join?”

### **F.8.5 What is the output record-list of a “ , ” Cross-Join?**

You may click:

- “F.8.5.1 What are the output field values of a “ , ” Cross-Join?”
- “F.8.5.2 What are the output records of a “ , ” Cross-Join?”
- “F.8.5.3 How many output records does a “ , ” Cross-Join produce?”

If you rather want to know what are the **output fields** of a “ , ” **Cross-Join**, you may click “F.8.4 What are the output fields of a Join?”.

#### **F.8.5.1 What are the output field values of a “ , ” Cross-Join?**

The **output** record produced by **joining** (click *F.8.1*) the record pair (**l**, **r**) has the field **values** (in the same order) from the **left “l”** record **followed by** the field **values** (in the same order) from the **right “r”** record.

#### **F.8.5.2 What are the output records of a “ , ” Cross-Join?**

A **Cross-Join** joins **all** (**l**, **r**) pairs.

In more detail, a **Cross-Join** joins (click *F.8.1*) **each and every “l” record** from its **left (first) “L” input** record-list with **each and every “r” record** from its **right (second) “R” input** record-list (this is called the “Cartesian product”).

The **output** record-list of a **Cross-Join** is therefore the **joining** (click *F.8.1*) of **all** ordered pairs of **input** records in the **set** of pairs (**L**, **R**).

The order of the **output** records is **unknown**.

A **Cross-Join** **does not** cause the pointers to the **Table fields** in its “**SELECT**” **expressions to become Null**, and therefore it **does not create Nulls** (but **Nulls** existing in the **input** records **will stay** in the corresponding **output** records).

Notice that the SQL codeword for the **Cross-Join** operator is just a comma (“ , ”).

The **Cross-Join** operator is also called the **Full-Join** operator.

If you want to see the differences between a **Cross-Join** and the other **Join** operations, you may click “F.8.2 What are the Join operators?”.

I am explaining **all** the **Join** operations with an example based on **the same two Tables**, so you can clearly see the differences between the results they produce. One Table is a list of house owners, and the other Table is a list of car owners. The field “**ID**” represents a **unique** person identification number (e.g., Social Security) to avoid mistaking two

different persons that have the same name. The example Tables are:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

Imagine that you want **all** ordered pairs of records between these two Tables: you can get this with a **Cross-Join**. The Query code<sup>44</sup> would be:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
    ,
    T_Car_owners AS Cars
```

The **output** record-list from this Query is:

F_Cross_Join					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	3	Peter Sellers	Renault Clio
14	Xi Liu	580 Beacon St.	3	Peter Sellers	Renault Clio
3	Peter Sellers	345 Bolton St.	6	Xi Liu	Ford Focus
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.	6	Xi Liu	Ford Focus
3	Peter Sellers	345 Bolton St.	18	John Welsh	Opel Zafira
6	Xi Liu	1342 Main St.	18	John Welsh	Opel Zafira
14	Xi Liu	580 Beacon St.	18	John Welsh	Opel Zafira
3	Peter Sellers	345 Bolton St.	23	Juan Perez	Ford Mustang
6	Xi Liu	1342 Main St.	23	Juan Perez	Ford Mustang
14	Xi Liu	580 Beacon St.	23	Juan Perez	Ford Mustang

Notice how the **duplicated** field names are **qualified** by **prefixing** each of them with the **name** of the **input** record-list that originated it, with an intermediate period “.” character (click C.2.2). The first two field names are prefixed with the **name** “Houses.”, while field names four and five are prefixed with the **name** “Cars.”. Notice also that in the SQL code above these two **names** were assigned to the **input** record-lists of the **Join** operation using the “**AS**” clause (click F.8.9).

<sup>44</sup> This is the Query “F\_Join\_Cross” from file “Company\_Database.accdb”..

The following picture **graphically** shows how the **output** records are produced:

Records from T_House_Owners	14-Xi Liu- 580 Beacon St.	14-Xi Liu- 580 Beacon St.- 3-Peter Sellers- Renault Clio	14-Xi Liu- 580 Beacon St.- 6-Xi Liu- Ford Focus	14-Xi Liu- 580 Beacon St.- 18-John Welsh- Opel Zafira	14-Xi Liu- 580 Beacon St.- 23-Juan Perez- Ford Mustang
	6-Xi Liu- 1342 Main St.	6-Xi Liu- 1342 Main St.- 3-Peter Sellers- Renault Clio	6-Xi Liu- 1342 Main St.- 6-Xi Liu- Ford Focus	6-Xi Liu- 1342 Main St.- 18-John Welsh- Opel Zafira	6-Xi Liu- 1342 Main St.- 23-Juan Perez- Ford Mustang
	3-Peter Sellers- 345 Bolton St.	3-Peter Sellers- 345 Bolton St.- 3-Peter Sellers- Renault Clio	3-Peter Sellers- 345 Bolton St.- 6-Xi Liu- Ford Focus	3-Peter Sellers- 345 Bolton St.- 18-John Welsh- Opel Zafira	3-Peter Sellers- 345 Bolton St.- 23-Juan Perez- Ford Mustang
		3-Peter Sellers- Renault Clio	6-Xi Liu- Ford Focus	18-John Welsh- Opel Zafira	23-Juan Perez- Ford Mustang
		Records from T_Car_Owners			

These **12 output** records were produced by **joining** the **3** records from the **left input** record-list with the **4** records from the **right input** record-list.

Notice that in this example I wrote the **Select** operation to produce **all** the fields, and in the **same order**, from its enclosed **Join** operation, but in the general case you can write the **Select** operation with whatever **output** fields, each with its specific expression, as you want.

If you want to know what are the **output fields** of “,” **Cross-Join**, you may click “F.8.4 What are the output fields of a Join?”.

**F.8.5.3 How many output records does a “,” Cross-Join produce?**

Knowing how many records a **Cross-Join** produces is **very** useful when **debugging** your Queries, and also for **better** understanding how a **Cross-Join** works.

A **Cross-Join** produces:

$$\text{num(L)} * \text{num(R)}$$

records, where “**L**” is the **left (first) input** record-list, and “**R**” is the **right (second) input** record-list.

The function “**num()**” represents the number of records in the record-list provided as its argument.

**F.8.6 What is the output record-list of an “INNER JOIN”?**

You may click:

- “F.8.6.1 What are the output field values of an “INNER JOIN”?”

- “F.8.6.2 What are the output records of an “INNER JOIN”?”
- “F.8.6.3 How many output records does an “INNER JOIN” produce?”

If you rather want to know what are the **output fields** of “INNER JOIN”, you may click “F.8.4 What are the output fields of a Join?”.

### F.8.6.1 What are the output field values of an “INNER JOIN”?

The **output** record produced by **joining** (click F.8.1) the **matching** record pair (**l<sub>m</sub>**, **r<sub>m</sub>**) has the field **values** (in the same order) from the **left** “**l<sub>m</sub>**” record **followed** by the field **values** (in the same order) from the **right** “**r<sub>m</sub>**” record.

Remind that a record pair (**l<sub>m</sub>**, **r<sub>m</sub>**) **matches** when the field **values** of “**l<sub>m</sub>**” and “**r<sub>m</sub>**” **jointly** produce **True** in the “**ON**” **Boolean** expression.

### F.8.6.2 What are the output records of an “INNER JOIN”?

An “INNER JOIN” joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs.

Remind that a record pair (**l<sub>m</sub>**, **r<sub>m</sub>**) **matches** when the field **values** of “**l<sub>m</sub>**” and “**r<sub>m</sub>**” **jointly** produce **True** in the “**ON**” **Boolean** expression.

In more detail, an “INNER JOIN” joins **every** “**l<sub>m</sub>**” **record** from its **left** (first) “**L**” **input** record-list with **every** “**r<sub>m</sub>**” **record** from its **right** (second) “**R**” **input** record-list **such that** the field **values** of “**l<sub>m</sub>**” and “**r<sub>m</sub>**” **jointly** return **True** in the “**ON**” **Boolean** expression.

An “INNER JOIN” **does not** cause the pointers to the **Table fields** in its “**SELECT**” **expressions to become Null**, and therefore it **does not create Nulls** (but **Nulls** existing in the **input** records **will stay** in the corresponding **output** records).

Notice that an “INNER JOIN” produces a **subset** of the **output** records produced by a **Cross-Join**. The **subset** corresponds to those **Cross-Join**’s **output** records arising from **matching input** records.

The order of the **output** records is **unknown**.

If you want to see the differences between a “INNER JOIN” and the other **Join** operations, you may click “F.8.2 What are the Join operators?”.

I am explaining **all** the **Join** operators with an example based on **the same two Tables**, so you can clearly see the differences between the results they produce. One Table is a list of house owners, and the other Table is a list of car owners. The field “**ID**” represents a **unique** person identification number (e.g., Social Security) to avoid mistaking two different persons that have the same name. The example Tables are:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

Imagine that you want a listing of persons that own a **house** and also own a **car**, indicating for each of them his/her **house address** and **car model**: you can get this with an “**INNER JOIN**”. The Query code<sup>45</sup> would be:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
INNER JOIN
    T_Car_owners AS Cars
ON Houses.ID = Cars.ID ;
```

The **output** record-list from this Query is:

F_Inner_Join_1					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus

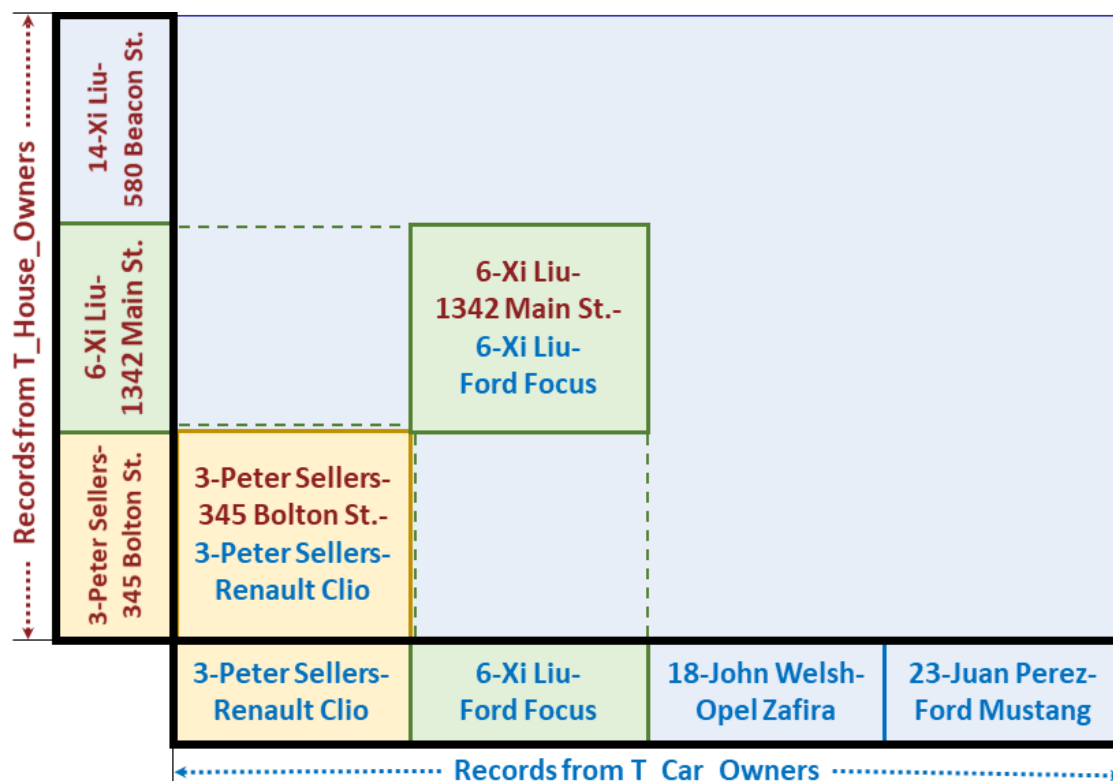
Notice how the **duplicated** field names are **qualified** by **prefixing** each of them with the **name** of the **input** record-list that originated it (and a separation period “.” character). The first two field names are prefixed with **name** “**Houses.**”, while field names four and five are prefixed with **name** “**Cars.**”. Notice also that these **names** were assigned to the **input** record-lists of the **Join** operation using the SQL clause “**AS**” (click *F.8.3*).

Notice how these two **output** records correspond to the only two pairs of **left** and **right input** records that match (i.e., produce **True**) in the “**ON**” **Boolean** expression.

---

<sup>45</sup> This is the Query “**F\_Join\_Inner\_1**” from file “**Company\_Database.accdb**”.

The following picture **graphically** shows how the **output** records are produced:



These **2 output** records were produced by **joining 1 matching** record from the **left input** record-list with **1 matching** record from the **right input** record-list, **plus, 1 matching** record from the **left input** record-list with **1 matching** record from the **right input** record-list. The colors of the term “**matching**” refer to the colors of the records in the picture.

Notice that in this example I wrote the **Select** operation to produce **all** the fields, and in the **same order**, from its enclosed **Join** operation, but in the general case you can write the **Select** operation with whatever **output** fields, each with its specific expression, as you want.

Notice also that it is **mandatory** that **each and every individual Boolean** expression within the “**ON**” **Boolean** expression contains **qualified** field names from **both input** record-lists. Notice also that the “**ON**” **Boolean** expression may produce **Null** and/or errors. If you want to know more about this, you may click “[F.8.9 What is the “ON” clause of “INNER JOIN” and Outer-Join \(“LEFT JOIN” and “RIGHT JOIN”\)?](#)”.

The example I just used may lead some SQL beginners to **wrongly believe** that in an “**INNER JOIN**”, **each** record from one of the **input** record-lists can **only match one or none** records from the other record-list. Actually, the matching of records is extremely versatile and depends on the specific “**ON**” **Boolean** expression that you write. Writing a suitable “**ON**” **Boolean** expression you can produce almost any **output** record-list that you want.

Let me show you this with an example. If we used the “**Name**” field instead of the “**ID**”

field for the “ON” matching of records, we would get the following Query code<sup>46</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
INNER JOIN
    T_Car_owners AS Cars
ON Houses.Name = Cars.Name ;
```

The **output** record-list from this Query is:

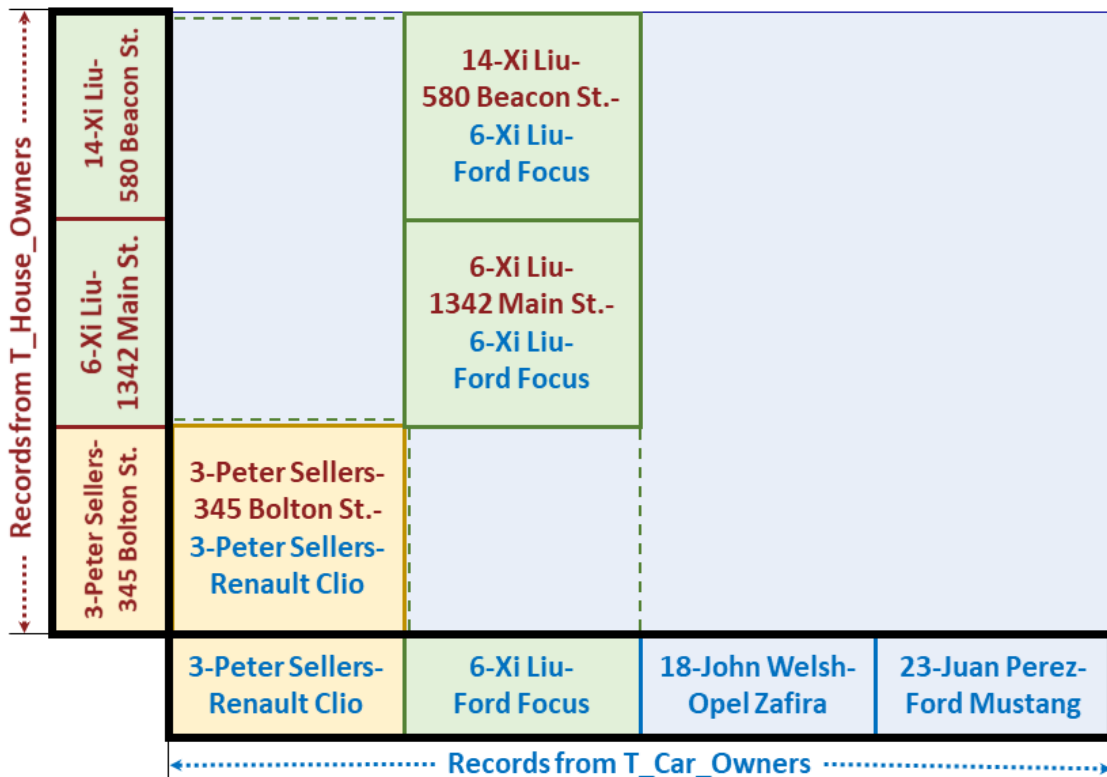
F_Inner_Join_2					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.	6	Xi Liu	Ford Focus

Notice this **output** record-list is **different** from the one of “INNER JOIN” in the previous example, where we used the “ON” expression “Houses.ID = Cars.ID”. In the present example, we got the **additional** record:

Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
14	Xi Liu	580 Beacon St.	6	Xi Liu	Ford Focus

which comes from the matching of **two different persons** having the **same name**.

The following picture **graphically** shows how the **output** records are produced:



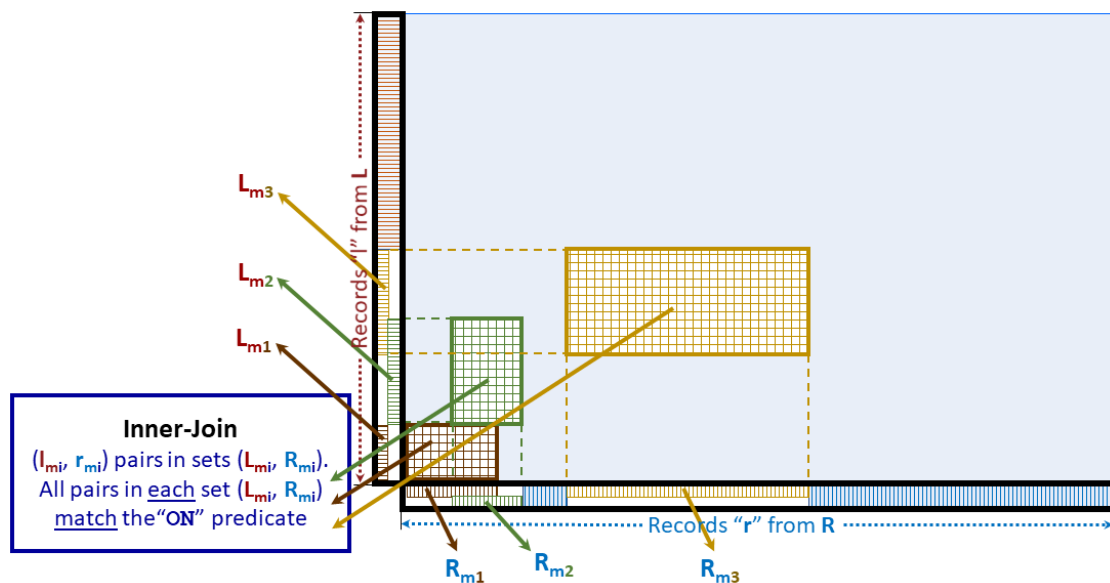
These **3 output** records were produced by **joining 1 matching** record from the **left input**

<sup>46</sup> This is the Query “F\_Join\_Inner\_2” from file “Company\_Database.accdb”.



record-list with **1 matching** record from the **right input** record-list, **plus**, **2 matching** records from the **left input** record-list with **1 matching** record from the **right input** record-list. The colors of the term “**matching**” refer to the colors of the records in the picture. You may see how the **two left input** records “**6-Xi Liu-1342 Main St.**” and “**14-Xi Liu-580 Beacon St.**” **match** with the **right input** record “**6-Xi Liu-Ford Focus**”.

The way an “**INNER JOIN**” works is that its “**ON**” *Boolean* expression will define **N matching record-list pairs** “**L<sub>mi</sub>**” and “**R<sub>mi</sub>**” such that **all** records in “**L<sub>mi</sub>**” **match** with **all** records in “**R<sub>mi</sub>**”. For **each** such “**i**” matching record-list pair, the “**INNER JOIN**” will join **all** the ordered record pairs (**l<sub>mi</sub>**, **r<sub>mi</sub>**) in each set (**L<sub>mi</sub>**, **R<sub>mi</sub>**), as if it were a **Cross-Join**. Since you have “**N**” such record-list pairs, the “**INNER JOIN**” will join all the ordered record pairs (**l<sub>mi</sub>**, **r<sub>mi</sub>**) in the sets (**L<sub>mi</sub>**, **R<sub>mi</sub>**) “**1**” to “**N**”. The following picture shows this **graphically**:



Notice in the picture how the **left matching** record-lists “**L<sub>mi</sub>**” may **overlap** among **themselves**, same as the **right matching** record-lists **R<sub>mi</sub>** may **overlap** among **themselves**. However, the rectangles (**L<sub>mi</sub>**, **R<sub>mi</sub>**) may **not** overlap among themselves.

The “**ON**” **expression** allows you to get the **joined** ordered **input** record pairs that you want in a **very flexible** way. To show an extreme case, you can write an “**ON**” *Boolean* expression such that the “**INNER JOIN**” produces **the same output** record-list as a **Cross-Join**. For example, if you run the following Query<sup>47</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
INNER JOIN
    T_Car_owners AS Cars
ON Houses.ID*Cars.ID*0 = 0 ;
```

you will see that its **output** record-list is the same as in a **Cross-Join**: if you check the “**ON**” *Boolean* expression you will quickly realize that it is **satisfied** by **each and every pair** of **input** records.

As an exercise, you may think of an “**ON**” *Boolean* expression that will produce **zero**

<sup>47</sup> This is the Query “**F\_Join\_Inner\_Cross**” from file “**Company\_Database.accdb**”.

**output** records in an “**INNER JOIN**” between the two example Tables.

A useful way to understand the “**INNER JOIN**” is thinking of it as a **Cross-Join** with a “**WHERE**” clause. The following two Queries produce **exactly the same output** record-list:

```

SELECT L.a, L.b, ..., L.k, R.a, R.b, ..., R.n
FROM
  L, R
WHERE Boolean-exp_X(L.a, L.b, ..., L.k, R.a, R.b, ..., R.n)

SELECT L.a, L.b, ..., L.k, R.a, R.b, ..., R.n
FROM
  L INNER JOIN R
ON Boolean-exp_X(L.a, L.b, ..., L.k, R.a, R.b, ..., R.n)

```

This being true, you may wonder...“*why do we need the “**INNER JOIN**” at all?*” The answer is that an **Inner-Join** operation is more efficient than its equivalent **Cross-Join** operation.

### F.8.6.3 How many output records does an “**INNER JOIN**” produce?

Knowing how many records an “**INNER JOIN**” produces is **very** useful when debugging your Queries, and also for **better** understanding how a “**INNER JOIN**” works.

An “**INNER JOIN**” produces:

$$\text{num}(\mathbf{L}_{m1}) * \text{num}(\mathbf{R}_{m1}) + \text{num}(\mathbf{L}_{m2}) * \text{num}(\mathbf{R}_{m2}) + \dots + \text{num}(\mathbf{L}_{mN}) * \text{num}(\mathbf{R}_{mN})$$

records, where each “**L<sub>mi</sub>**” is a subset of “**L**” and each “**R<sub>mi</sub>**” is a subset “**R**”, such that **all** records in “**L<sub>mi</sub>**” **match** with **all** records in “**R<sub>mi</sub>**”. The **index “i”** goes from **1** to “**N**”. It is also required that the “**N**” (**L<sub>mi</sub>**, **R<sub>mi</sub>**) sets of record pairs contain **exactly the same** record pairs as all the **matching record pairs** (**l<sub>m</sub>**, **r<sub>m</sub>**) between “**L**” and “**R**”.

The function “**num()**” represents the number of records in the record-list provided as its argument.

### F.8.7 What is the output record-list of an Outer-Join (“**LEFT JOIN**” or “**RIGHT JOIN**”)?

You may click:

- “F.8.7.1 What are the output field values of an Outer-Join (“**LEFT JOIN**” or “**RIGHT JOIN**”)?”
- “F.8.7.2 What are the output records of an Outer-Join (“**LEFT JOIN**” or “**RIGHT JOIN**”)?”
- “F.8.7.3 How many output records does an Outer-Join (“**LEFT JOIN**” or “**RIGHT JOIN**”) produce?”

If you rather want to know what are the **output fields** of an **Outer-Join**, you may click “F.8.4 What are the output fields of a Join?”.

### F.8.7.1 What are the output field values of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?

For both “LEFT JOIN” and “RIGHT JOIN” the **output** record produced by **joining** (click *F.8.1*) the **matching** record pair ( $\mathbf{l}_m, \mathbf{r}_m$ ) has the field **values** (in the same order) from the **left** “ $\mathbf{l}_m$ ” record **followed by** the field **values** (in the same order) from the **right** “ $\mathbf{r}_m$ ” record.

Remind that a record pair ( $\mathbf{l}_m, \mathbf{r}_m$ ) **matches** when the field **values** of “ $\mathbf{l}_m$ ” and “ $\mathbf{r}_m$ ” **jointly** produce *True* in the “ON” *Boolean* expression.

For a “LEFT JOIN”, the **output** record produced by **joining** (click *F.8.1*) the **non-matching** record pair ( $\mathbf{l}_{nm}, \mathbf{void}$ ) has the field **values** (in the same order) from the **left** “ $\mathbf{l}_{nm}$ ” record **followed by** the “SELECT” **expressions** of the **right input** record-list “R” with **Null** pointers to **all** their Table fields.

For a “RIGHT JOIN”, the **output** record produced by **joining** (click *F.8.1*) the **non-matching** record pair ( $\mathbf{void}, \mathbf{r}_{nm}$ ) has “SELECT” **expressions** of the **left input** record-list “L” will **Null** pointers to **all** their Table fields **followed by** the field **values** (in the same order) from the **right** “ $\mathbf{r}_{nm}$ ” record.

The **Null** pointers to Table fields that are **created** in **void** fields may cause a “SELECT” **expression** to **crash**, or to return an **exception-value** or a **Null**. You should therefore **carefully** handle these created **Null** pointers (you may click “K.5 Why and how should I carefully handle Nulls in my Queries?”).

### F.8.7.2 What are the output records of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?

A “LEFT JOIN” joins **all** ( $\mathbf{l}_m, \mathbf{r}_m$ ) **matching** pairs, **plus**, **all** the ( $\mathbf{l}_{nm}, \mathbf{void}$ ) **non-matching** pairs.

We say that that ( $\mathbf{l}_{nm}, \mathbf{void}$ ) is a **non-matching** pair when **all** pairs in the set ( $\mathbf{l}_{nm}, \mathbf{R}$ ) **do not match**. This is, when the “ $\mathbf{l}_{nm}$ ” record does not match with **any** record in “R”.

In more detail, a “LEFT JOIN” joins the same ( $\mathbf{l}_m, \mathbf{r}_m$ ) **input** record ordered pairs as an “INNER JOIN”, **plus**, all the ordered record pairs ( $\mathbf{l}_{nm}, \mathbf{void}$ ) in the set ( $\mathbf{L}_{nm}, \mathbf{void}$ ).

In this statement, the record-list “ $\mathbf{L}_{nm}$ ” contains **all** the **non-matching** “ $\mathbf{l}_{nm}$ ” **input** records. This is, “ $\mathbf{L}_{nm}$ ” is the **subset** of “L” such that **all** the pairs ( $\mathbf{l}_{nm}, \mathbf{R}$ ) **do not match**.

A **void** record has **all** its fields **void**. A **void** field has **Null** pointers to **all** the Table fields in its **expanded** “SELECT” **expression**.

A “RIGHT JOIN” is exactly the same as a “LEFT JOIN”, with the only difference that the **input** record-list that produces the additional set of records, is the **right** (**second**) one instead of the **left** (**first**) one. Therefore, the **Null** pointers to Table fields that are **created** appear in the “SELECT” **expressions** of the **left** (**first**) “L” **input** record-list.

Therefore, a “RIGHT JOIN” joins **all** ( $\mathbf{l}_m, \mathbf{r}_m$ ) **matching** pairs, **plus**, **all** the ( $\mathbf{void}, \mathbf{r}_{nm}$ ) **non-matching** pairs.

We say that that ( $\mathbf{void}, \mathbf{r}_{nm}$ ) is a **non-matching** pair when **all** pairs in the set ( $\mathbf{L}, \mathbf{r}_{nm}$ ) **do not match**. This is, when the “ $\mathbf{r}_{nm}$ ” record does not match with **any** record in “L”.

Using “LEFT JOIN” is usually preferable to using “RIGHT JOIN” because it is

considered that the resulting SQL code is more readable. However, a “**RIGHT JOIN**” is also useful on some cases.

For both “**LEFT JOIN**” and “**RIGHT JOIN**” the order of the **output** records is **unknown**.

If you want to see the differences between **Outer-Join** operators and the other **Join** operations, you may click “*F.8.2 What are the Join operators?*”.

I am explaining **all** the **Join** operators with an example based on **the same two Tables**, so you can clearly see the differences between the results they produce. One Table is a list of house owners and the other Table is a list of car owners. The field “**ID**” represents a **unique** person identification number (e.g., Social Security) to avoid mistaking two different persons that have the same name. The example Tables are:

T_House_owners			T_Car_owners		
ID	Name	Address	ID	Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.	18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

Imagine that you want a listing of **all** persons that own a **house**, **also indicating their car model** (in case they own one). You can get this with a “**LEFT JOIN**”. The Query code<sup>48</sup> would be:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
LEFT JOIN
    T_Car_owners AS Cars
ON Houses.ID = Cars.ID ;
```

The **output** record-list from this Query is:

F_Left_Join_1					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.			

Notice how the **duplicated** field names are **qualified** by **prefixing** each of them with the **name** of the **input** record-list that originated it (and a separation period “.” character). The first two field names are prefixed with “**Houses.**”, while field names four and five are prefixed with “**Cars.**”. Notice also that these names were assigned to the **input** record-lists of the **Join** operation using the SQL clause “**AS**” (click *F.8.3*).

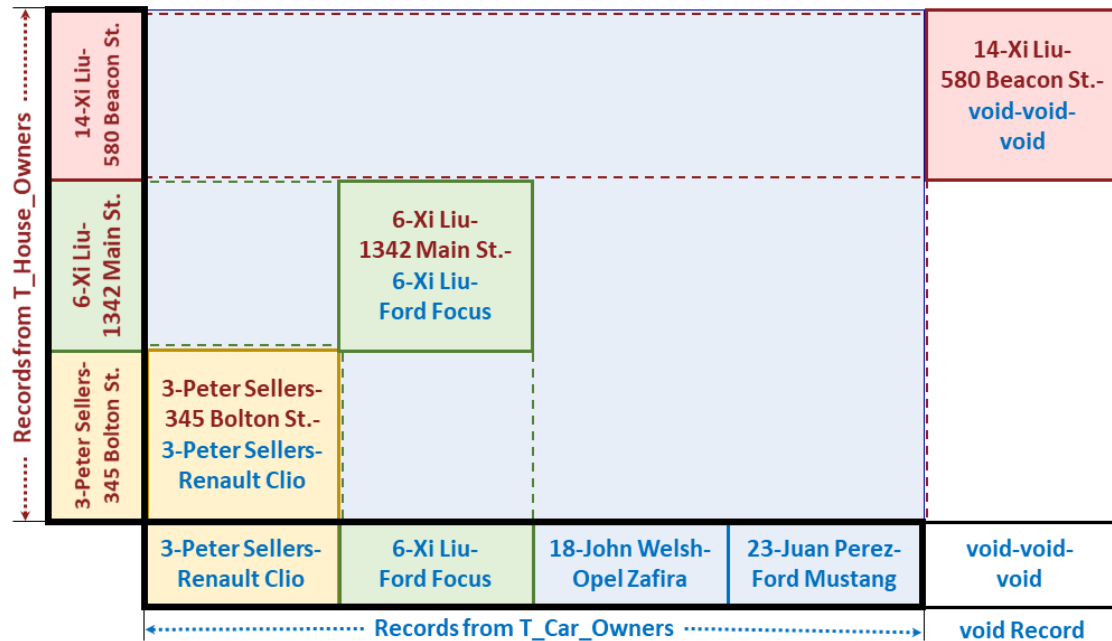
Notice how the **output** records correspond to the ones of an “**INNER JOIN**”, **plus**, the **additional** record:

<sup>48</sup> This is the Query “**F\_Join\_Left\_1**” from file “**Company\_Database.accdb**”.

Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
14	Xi Liu	580 Beacon St.			

This **additional output** record arises from joining the only **non-matching left input** record “14-Xi Liu-580 Beacon St.” with a **void right** record.

The following picture **graphically** shows how the **output** records are produced:



These **3 output** records were produced by **joining 1 matching** record from the **left input** record-list with **1 matching** record from the **right input** record-list, **plus, 1 matching** record from the **left input** record-list with **1 matching** record from the **right input** record-list, **plus, 1 non-matching** record from the **left input** record-list with a **void right** record. The colors of the term “**matching**” refer to the colors of the records in the picture.

Based on the picture above, a trick to remind the difference between the **Inner-Join** and the **Outer-Joins** is the following:

- **Inner-Join** **only** produces records **inside** the ones of the Cartesian product (i.e., **inside** the ones of a **Cross-Join**).
- **Outer-Joins** **may** produce records **outside** the ones of the Cartesian product (i.e., **outside** the ones of a **Cross-Join**).

Notice that in this example I wrote the **Select** operation to produce **all** the fields, and in the **same order**, from its enclosed **Join** operation, but in the general case you can write the **Select** operation with whatever **output** fields, each with its specific expression, as you want.

Notice also that it is **mandatory** that **each and every individual Boolean** expression within the “**ON**” **Boolean** expression contains **qualified** field names from **both input** record-lists. Notice also that the “**ON**” **Boolean** expression may produce **Null** and/or errors. If you want to know more about this, you may click “*F.8.9 What is the “ON” clause of “INNER JOIN” and Outer-Join (“LEFT JOIN” and “RIGHT JOIN”)?*”.

As another example, in case you wanted **instead** a listing of **all** persons that own a **car**, **also indicating their home address** (in case they own a house). You can get this by just **exchanging** the two **input** record-lists in the “**LEFT JOIN**”. However, notice that **I am not exchanging the output field names** from the enclosing “**SELECT**” statement (so the left/right colors are exchanged), nor the field names in the “**ON**” expression. The resulting Query is<sup>49</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
  T_Car_owners AS Cars
LEFT JOIN
  T_House_owners AS Houses
ON Houses.ID = Cars.ID ;
```

The resulting **output** record-list is:

F_Join_Left_2					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
			18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

Notice how the **output** records correspond to the ones of an “**INNER JOIN**”, **plus**, the two **additional** records:

Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
			18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

These two **additional output** records arise from joining the **2 non-matching left input** records “**18-John Welsh-Opel Zafira**” and “**23-Juan Perez-Ford Mustang**” with a **void right** record. Notice that the “**SELECT**” is **changing the field order**. Remind that Null fields are displayed in MS-Access as **blank** fields.

### F.8.7.3 How many output records does an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”) produce?

Knowing how many records an **Outer-Join** produces is **very** useful when debugging your Queries, and also for **better** understanding how “**LEFT JOIN**” and “**RIGHT JOIN**” work.

A “**LEFT JOIN**” produces:

$$\text{num}(L_{m1}) * \text{num}(R_{m1}) + \dots + \text{num}(L_{mN}) * \text{num}(R_{mN}) + \text{num}(L_{nm})$$

record.

A “**RIGHT JOIN**” produces:

$$\text{num}(L_{m1}) * \text{num}(R_{m1}) + \dots + \text{num}(L_{mN}) * \text{num}(R_{mN}) + \text{num}(R_{nm})$$

<sup>49</sup> This is the Query “F\_Join\_Left\_1” from file “Company\_Database.accdb”.



records.

Same as in “**INNER JOIN**”, each “**L<sub>mi</sub>**” is a subset of “**L**” and each “**R<sub>mi</sub>**” is a subset of “**R**”, such that **all** records in “**L<sub>mi</sub>**” **match** with **all** records in “**R<sub>mi</sub>**”. The **index** “**i**” goes from **1** to “**N**”. It is also required that the “**N**” (**L<sub>mi</sub>**, **R<sub>mi</sub>**) **sets of record pairs** contain **exactly the same** record pairs as all the **matching record pairs** (**l<sub>m</sub>**, **r<sub>m</sub>**) between “**L**” and “**R**”.

The **non-matching left** record-list “**L<sub>nm</sub>**” includes all the non-matching records “**l<sub>nm</sub>**” that **do not match** with **any** record in “**R**”.

The **non-matching right** record-list “**R<sub>nm</sub>**” includes all the non-matching records “**r<sub>nm</sub>**” that **do not match** with **any** record in “**L**”.

The function “**num()**” represents the number of records in the record-list provided as its argument.

### **F.8.8 What is the output record-list of a Full-Outer-Join?**

You may click:

- “*F.8.8.1 What are the output field values of a Full-Outer-Join?*”
- “*F.8.8.2 What are the output records of a Full-Outer-Join?*”
- “*F.8.8.3 How many output records does a Full-Outer-Join produce?*”

If you rather want to know what are the **output fields** of a **Full-Outer-Join**, you may click “*F.8.4 What are the output fields of a Join?*”.

#### **F.8.8.1 What are the output field values of a Full-Outer-Join?**

The **output** record produced by **joining** (click *F.8.1*) the **matching** record pair (**l<sub>m</sub>**, **r<sub>m</sub>**) has the field **values** (in the same order) from the **left** “**l<sub>m</sub>**” record **followed by** the field **values** (in the same order) from the **right** “**r<sub>m</sub>**” record.

Remind that a record pair (**l<sub>m</sub>**, **r<sub>m</sub>**) **matches** when the field **values** of “**l<sub>m</sub>**” and “**r<sub>m</sub>**” **jointly** produce **True** in the “**ON**” **Boolean** expression.

The **output** record produced by **joining** (click *F.8.1*) the **non-matching** record pair (**l<sub>nm</sub>**, **void**) has the field **values** (in the same order) from the **left** “**l<sub>nm</sub>**” record **followed by** the “**SELECT**” **expressions** of the **right input** record-list “**R**” with **Null** pointers to **all** their **Table fields**.

The **output** record produced by **joining** (click *F.8.1*) the **non-matching** record pair (**void**, **r<sub>nm</sub>**) has “**SELECT**” **expressions** of the **left input** record-list “**L**” will **Null** pointers to **all** their **Table fields** **followed by** the field **values** (in the same order) from the **right** “**r<sub>nm</sub>**” record.

The **Null** pointers to **Table fields** that are **created in void** fields may cause a “**SELECT**” **expression** to **crash**, or to return an **exception-value** or a **Null**. You should therefore **carefully** handle these created **Null** pointers (you may click “*K.5 Why and how should I carefully handle Nulls in my Queries?*”).

#### **F.8.8.2 What are the output records of a Full-Outer-Join?**

A **Full-Outer-Join** joins **all** (**l<sub>m</sub>**, **r<sub>m</sub>**) **matching** pairs, **plus**, **all** the (**l<sub>nm</sub>**, **void**) and **all** the (**void**, **r<sub>nm</sub>**) **non-matching** pairs.



We say that that (**l<sub>nm</sub>**, **void**) is a **non-matching** pair when **all** pairs in the set (**l<sub>nm</sub>**, **R**) **do not match**. This is, when the “**l<sub>nm</sub>**” record does not match with **any** record in “**R**”.

We say that that (**void**, **r<sub>nm</sub>**) is a **non-matching** pair when **all** pairs in the set (**L**, **r<sub>nm</sub>**) **do not match**. This is, when the “**r<sub>nm</sub>**” record does not match with **any** record in “**L**”.

In other words, a **Full-Outer-Join** joins the same pairs as an “**INNER JOIN**”, **plus**, all the **additional** (**l<sub>nm</sub>**, **void**) pairs of a “**LEFT JOIN**”, **plus**, all the **additional** (**void**, **r<sub>nm</sub>**) pairs of a “**RIGHT JOIN**”.

A **void** record has **all** its fields **void**. A **void** field has **Null** pointers to **all** the **Table fields** in its expanded “**SELECT**” **expression**.

The order of the **output** records is **unknown**.

MS-Access **does not provide** an operator to produce a **Full-Outer-Join**. However, I am explaining **Full-Outer-Join** because it is an important concept, and on some cases you may need to write an SQL piece of code for a **Full-Outer-Join**. If you want to know more about the **output** record-list of a **Full-Outer-Join**, you may click “*F.8.8 What is the output record-list of a Full-Outer-Join?*”. If you want to know how to code a **Full-Outer-Join**, go the end of this section.

If you want to see the differences between a **Full-Outer-Join** and the other **Join** operations, you may click “*F.8.2 What are the Join operators?*”.

I am explaining **all** the **Join** operators with an example based on **the same two Tables**, so you can clearly see the differences between the results they produce. One Table is a list of house owners and the other Table is a list of car owners. The field “**ID**” represents a **unique** person identification number (e.g., Social Security) to avoid mistaking two different persons that have the same name. The example Tables are:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

Imagine that you want a listing of persons that own a **house** **or** own a **car**, **indicating** their **house address** (in case they own one) **and** their **car model** (in case they own one). You can get this with a **Full-Outer-Join**. In case the “**Full-Outer-Join**” operator existed, the Query code<sup>50</sup> would be:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
  T_House_owners AS Houses
Full-Outer-Join
  T_Car_owners AS Cars
ON Houses.ID = Cars.ID ;
```

<sup>50</sup> This is the Query “**F\_Join\_Full\_Outer**” from file “**Company\_Database.accdb**”.

The **output** record-list from such a Query would be:

F_Full_Outer_Join					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.			
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
			18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

Notice how the **duplicated** field names are **qualified** by **prefixing** each of them with the **name** of the **input** record-list that originated it (and a separation period “.” character). The first two field names are prefixed with “**Houses.**”, while field names four and five are prefixed with “**Cars.**”. Notice also that these names were assigned to the **input** record-lists of the **Join** operation using the SQL clause “**AS**” (click *F.8.3*).

Notice how the **output** records correspond to the ones of an “**INNER JOIN**”, **plus**, the three **additional** records:

Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
14	Xi Liu	580 Beacon St.			
			18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

The first **additional output** record arises from joining the **1 non-matching left input** record “**14-Xi Liu-580 Beacon St.**” with a **void right** record.

The last **2 additional output** records arise from joining a **void left** record with the **2 non-matching right input** records “**18-John Welsh-Opel Zafira**” and “**23-Juan Perez-Ford Mustang**”.

Remind that **Null** fields are displayed in MS-Access as **blank** fields.



### F.8.8.3 How many output records does a Full-Outer-Join produce?

Knowing how many records a **Full-Outer-Join** produces is **very** useful when debugging your Queries, and also for **better** understanding how a **Full-Outer-Join** works.

A **Full-Outer-Join** produces:

$$\text{num}(\mathbf{L}_{m1}) * \text{num}(\mathbf{R}_{m1}) + \dots + \text{num}(\mathbf{L}_{mN}) * \text{num}(\mathbf{R}_{mN}) + \text{num}(\mathbf{L}_{nm}) + \text{num}(\mathbf{R}_{nm})$$

records.

Same as in “**INNER JOIN**”, each “**L<sub>mi</sub>**” is a **subset** of “**L**” and each “**R<sub>mi</sub>**” is a **subset** “**R**”, such that **all** records in “**L<sub>mi</sub>**” **match** with **all** records in “**R<sub>mi</sub>**”. The **index** “**i**” goes from **1** to “**N**”. It is also required that the “**N**” (**L<sub>mi</sub>**, **R<sub>mi</sub>**) **sets of record pairs** contain **exactly the same** record pairs as all the **matching record pairs** (**l<sub>m</sub>**, **r<sub>m</sub>**) between “**L**” and “**R**”.

The **non-matching left** record-list “**L<sub>nm</sub>**” includes all the non-matching records “**l<sub>nm</sub>**” that **do not match** with **any** record in “**R**”.

The **non-matching right** record-list “**R<sub>nm</sub>**” includes all the non-matching records “**r<sub>nm</sub>**” that **do not match** with **any** record in “**L**”.

The function “**num()**” represents the number of records in the record-list provided as its argument.

### F.8.9 What is the “ON” clause of “INNER JOIN” and Outer-Join (“LEFT JOIN” and “RIGHT JOIN”)?

In an “**INNER JOIN**”, “**LEFT JOIN**” or “**RIGHT JOIN**” operation, the mandatory “**ON**” clause indicates what are the **left** and **right matching** records, as follows:

```
ON ON-Boolean-exp (Ind-ON-Boolean-exps (Left_name.x, Right_name.y, ...))
```

The “**ON**” clause contains the “**ON**” keyword, followed by a **Boolean-expression** (denoted “**ON-Boolean-exp ()**” in the code above). This “**ON**” **Boolean-expression** is applied to the **joint values** of **each and every pair of left** and **right input** records of the **Join** operator, to determine if the pair **matches**.

If the result of the “**ON**” **Boolean** expression is **True**, then the record pair **matches**, and an **output** record is produced by **joining** the **left** and **right input** records in the pair.

If the result of the “**ON**” **Boolean** expression is **False** or **Null**, then the record pair **does not match**, and it **does not** produce an **output** record.

If the “**ON**” **Boolean-expression** produces an exception value (e.g., divide by zero), the Query will on most cases **crash**.

It is **mandatory** that the “**ON**” **Boolean** expression contains **at least one individual Boolean** expression that includes **at least one qualified** field name from **each** of the two **input** record-lists, such that if this **Boolean** expression is **True**, then the overall “**ON**” **Boolean** expression is also **True**. If this is not fulfilled, MS-Access will not allow you to save the Query code, and will typically present the error message:

“*JOIN expression not supported.*”

Remind that a **qualified** field name (click C.2.2) is a field name **prefixed** by the name

of the **input** record-list where it belongs with an intermediate period “.” character (e.g., “T\_House\_owners.Address”).

The above rule for an “**ON**” *Boolean* expression is not easy to apply, so my advice is that to avoid problems you **qualify all** the field names that you use in the “**ON**” *Boolean* expression. If MS-Access rejects an “**ON**” *Boolean* expression where **all** field names are **qualified**, this is because you are using field names from **only one** of the two **input** record-lists, and therefore this expression has to be taken to an inner “**WHERE**” clause (see further below).

To clarify the rule to write an “**ON**” *Boolean* expression that I have indicated above, check the following examples<sup>51</sup> over the two Tables from subsection F.8.2.2 (and other sections).

Four examples of **correct** “**ON**” *Boolean* expressions are:

```
ON (T_House_Owners.ID > T_Car_Owners.ID)
ON (T_House_Owners.ID*Len(Address) > T_Car_Owners.ID)
ON (T_House_Owners.ID > T_Car_Owners.ID )
  AND (T_House_Owners.Name <> T_Car_Owners.Name)
-- This one is correct, but NOT advisable, because not all field names have
-- been qualified
ON (T_House_Owners.ID > T_Car_Owners.ID )
  OR ( Address <> Car)
```

Three examples of **wrong** “**ON**” *Boolean* expressions are:

```
-- The "Right" field name is not qualified
ON (Len(T_House_Owners.ID) > Len(Car))
-- The second expression does not have a "Right" field name
ON (T_House_Owners.ID > T_Car_Owners.ID)
  AND (T_House_Owners.ID < 30)
-- The second expression has unqualified field names
ON (T_House_Owners.ID > T_Car_Owners.ID )
  AND ( Address <> Car)
```

Notice how the **last** correct (but **not advisable**) expression is **almost** the same as the **last** wrong one, with the **only difference** that the correct one uses “**OR**” while the wrong one uses “**AND**”.

The main reason why the “**ON**” *Boolean* expression **must** contain field names from **both** **input** record-lists is **efficiency**. If you want to set a given condition (e.g., a *Boolean* expression) over field names of **only one** of the **input** record-lists, it is much more efficient to do it using a “**WHERE**” clause over the corresponding **input** record-list. Therefore, an **individual Boolean** expression in an “**ON**” clause only makes sense if it involves field names from **both** **input** record-lists.

To summarize these writing rules for the “**ON**” *Boolean* expression, in the SQL syntax at the beginning of this section I have indicated that the **elements** used to build each “**Ind-ON-Boolean-exp()**” **must** include “**Left\_name.x**” and “**Right\_name.y**”, where “**Left\_name**” and “**Right\_name**” represent the names of the **left** and **right** **input** record-list, and “**x**” and “**y**” represent field names from the **left** and **right** **input** record-lists, respectively.

Changing topic, notice that a given “**ON**” *Boolean* expression may be **commutative** or

---

<sup>51</sup> You can find them in Query “F\_Join\_ON\_exp” in “Company\_Database.accdb”.

**not**. If the expression is **commutative**, then you **can exchange** the qualified field names from both **input** record-lists, and the Query result will be **the same**. If the expression is **not commutative** and you **exchange** the qualified field names from both **input** record-lists, then the Query result will be **different**. This is related to the question discussed in “F.8.2.2 *Are Join operations commutative?*”.

A couple examples of **commutative** and **non-commutative** “**ON**” *Boolean*-expressions are:

```
ON (T_House_Owners.ID = T_Car_Owners.ID) -- Commutative
ON (T_House_Owners.ID < T_Car_Owners.ID) -- Non-commutative
```

The “**ON**” clause **cannot** be used with the **Cross-Join** operator, and it **must** be used with the “**INNER JOIN**”, “**LEFT JOIN**” and “**RIGHT JOIN**” operators.

If you want to know what is the **output** record-list of a **Join** operator using the “**ON**” clause, you may click:

- “F.8.6 *What is the output record-list of an “INNER JOIN”?*”
- “F.8.7 *What is the output record-list of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?*”
- “F.8.8 *What is the output record-list of a Full-Outer-Join?*”

If you want to write (syntax) a correct “**ON**” clause, you may click F.8.10.

## F.8.10 How do I write a correct (syntax) Join?

You may click:

- “F.8.10.1 *What is a syntax-example of a Join?*”
- “F.8.10.2 *What are the formal rules (syntax) to write a Join?*”
- “F.8.10.3 *How do I nest Joins?*”

### F.8.10.1 What is a syntax-example of a Join?

An illustrative example of a fairly complete **Inner-Join operation**<sup>52</sup> is:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
INNER JOIN
    T_Car_owners AS Cars
ON (Houses.Name = Cars.Name) AND (Houses.ID <> Cars.ID)
```

A **Join** operation may not be the outermost operation of a Query. For this reason, the **Join** operation in this example is enclosed in a **Select** operation.

The **Join** operator in the example is “**INNER JOIN**”.

The **two input** record-lists (at both sides of the **Join** operator) are:

- The **left input** record-list is the Table name “**T\_House\_owners**”, to which I have assigned the name “**Houses**” using the optional “**AS**” clause.
- The **right input** record-list is the Table name “**T\_Car\_owners**”, to which I have

<sup>52</sup> This is the Query “F\_Join\_Syntax” from the “Company\_Database.accdb” file.



assigned the name “**Cars**” using the optional “**AS**” clause.

There are **two individual Boolean-expressions** in the “**ON**” **Boolean expression**:

- “**Houses.Name = Cars.Name**”, which (as required) includes a qualified field name from both the **left** and the **right input** record-lists.
- “**Houses.ID <> Cars.ID**”, which (as required) includes a qualified field name from both the **left** and the **right input** record-lists.
- The two **individual Boolean-expressions** are connected with the “**AND**” **Boolean operator**.
- Notice that if you remove **any** of the four qualifiers (i.e., one of the prefixes “**Houses.**” or “**Cars.**”) in the “**ON**” **Boolean expression**, you will not be able to save the Query and you will get a syntax error message.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

### F.8.10.2 What are the formal rules (syntax) to write a Join?

A correct **Join operation** has to be written in the following way:

```
{ [ [ (] {Table-name or Query-name} [ ] ]
or [ {Table-name or Query-name} [AS Left-name] ]
or [ ( {Select-opr or Union-opr } ) AS Left-name ]
or [ [ (] Inner-or-Outer-Join-opr [ ] ] or Cross-Join-opr ] }
{ , or INNER JOIN or LEFT [OUTER] JOIN or RIGHT [OUTER] JOIN }
{ [ [ (] {Table-name or Query-name} [ ] ]
or [ {Table-name or Query-name} [AS Right-name] ]
or [ ( {Select-opr or Union-opr } ) AS Right-name ]
or [ [ (] Inner-or-Outer-Join-opr [ ] ] or Cross-Join-opr ] }
[ ON ON-Boolean-exp (Ind-ON-Boolean-exps (Left_name.x, Right_name.y, ...)) ]
```

Right after the green curly brace “{” you may see a bold comma “,” and some words in **bold font**: all of them are SQL **keywords**. The parentheses enclosed in square brackets “[ ]” are optional. The elements separated with “or” are alternative options. Each list of alternative options for an element is enclosed between curly braces “{ }” when the element is **not optional** and between square brackets “[ ]” when the element is **optional**. Some curly braces “{ }” and square brackets “[ ]” are colored just to make it easier to see which ones are paired.

All the terms above with a trailing “-opr” are SQL operations.

The elements in gray text are the ones I advise you do not use:

- I advise you **do not use** the optional “**OUTER**” **keyword** just for conciseness, because this keyword does not add any semantics. The optional “**OUTER**” **keyword** can be placed in the middle of “**LEFT JOIN**” or “**RIGHT JOIN**”, as you may see above.
- I advise you **do not use** nested **Join** operations (except for the “,” **Cross-Join** operator) because it makes your SQL code far less readable.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

I will now explain how to write the two parts of a **Join** operation.



### How do I write the two input record-lists?

The two **input** record-list of the **Join** operator are written at **both** sides of the **Join** operator. The **left (first) input** record-list is written just before the **Join** operator, and the **right (second) input** record-list is written just after it. I now explain more detail about this.

Each of the two **input** record-lists can be written as either a **Table** name, a **Query** name, a **Select operation** or a **Union operation**. The two **input** record-lists **may be the same record-list**.

Notice that I have shaded above the **Inner-Join** and **Outer-Join** operations in gray text because **I advise you do not use them as an input record-list of another Join**. If you use either of them as the **input** record-list, the code becomes less readable and parentheses rules become complex: these are reasons why I advise you do not use them. If you want to do a **Join** operation over an **Inner** or **Outer Join** (which is quite frequent), just enclose the **Inner** or **Outer Join** in a **Select operation**. Notice though that you can nest **Cross-Joins** (i.e., do a **Cross-Join** of a **Cross-Join**) because the **Cross-Join** operation is associative and is not enclosed between parentheses.

The rules for **parentheses** and the “**AS**” clause depend on what is the **input record-list** after the “**FROM**” clause, as follows:

- **A Table name or Query name**  
It **may** be enclosed between parentheses, or, it may have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one of **Select** and **Transform** operations.
- **A Select operation or Union operation**  
It **must** be enclosed between parentheses and at least one of both **must** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the one of **Select**, **Transform** and **Union** operations.
- **A Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of **Select** and **Transform** operations.
- **A Cross-Join operation**  
It **must not** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one of **Select** and **Transform** operations.

### How do I write the “ON” clause?

The “**ON**” clause is **not** optional: it **cannot** be used with the **Cross-Join** operator, and it **must** be used with the “**INNER JOIN**”, “**LEFT JOIN**” and “**RIGHT JOIN**” operators. If you want to know more about the “**ON**” clause, you may click “[F.8.9 What is the “ON” clause of “INNER JOIN” and Outer-Join \(“LEFT JOIN” and “RIGHT JOIN”\)?”](#)”.

It is **mandatory** that the “**ON**” **Boolean** expression contains **at least one Boolean** expression that includes **at least one qualified** field name from **each** of the two **input** record-lists, such that if this **Boolean** expression is **True**, then the overall “**ON**” **Boolean**

expression is also *True*. If this is not fulfilled, MS-Access will not allow you to save the Query code, and will typically present the error message:

*“JOIN expression not supported.”*

Remind that a **qualified** field name (click C.2.2) is a field name **prefixed** by the name of the **input** record-list where it belongs with an intermediate period “.” character (e.g., “T\_House\_owners.Address”).

The above rule for an “**ON**” *Boolean* expression is not easy to apply, so my advice is that to avoid problems you **qualify all** the field names that you use in the “**ON**” *Boolean* expression. If MS-Access rejects an “**ON**” *Boolean* expression where **all** field names are **qualified**, this is because you are using field names from **only one** of the two **input** record-lists, and therefore this expression has to be taken to an inner “**WHERE**” clause.

To clarify the rule to write an “**ON**” *Boolean* expression I have indicated above, check the following examples<sup>53</sup> over the two Tables from subsection F.8.2.2.

Four examples of **correct** “**ON**” *Boolean* expressions are:

```
ON (T_House_Owners.ID > T_Car_Owners.ID)
ON (T_House_Owners.ID*Len(Address) > T_Car_Owners.ID)
ON (T_House_Owners.ID > T_Car_Owners.ID )
  AND (T_House_Owners.Name <> T_Car_Owners.Name)
-- This one is correct, but NOT advisable, because not all field names have
-- been qualified
ON (T_House_Owners.ID > T_Car_Owners.ID )
  OR ( Address <> Car)
```

Three examples of **wrong** “**ON**” *Boolean* expressions are:

```
-- The "right" field name is not qualified
ON (Len(T_House_Owners.ID) > Len(Car))
-- The second expression does not have a "right" field name
ON (T_House_Owners.ID > T_Car_Owners.ID)
  AND (T_House_Owners.ID < 30)
-- The second expression has unqualified field names
ON (T_House_Owners.ID > T_Car_Owners.ID )
  AND ( Address <> Car)
```

Notice how the **last** correct (but **not advisable**) expression is **almost** the same as the **last** wrong one, with the **only difference** that the correct one uses “**OR**” while the wrong one uses “**AND**”.

### F.8.10.3 How do I nest Joins?

Although you can actually use a **Join** operation as an **input** record-list of the **Join** operator (this is, **nesting Joins**), the code readability becomes **poor**, and it is **very easy** to make a **mistake** with the **parentheses**. For these reasons, I advise you **avoid nesting Join operations**. If you want to do a **Join** of a **Join**, just enclose the innermost join in a **Select operation**. As an exception to this advice, you can do a **Join** of a **Join** if both operators are a **Cross-Join**, because the **Cross-Join** operator is associative, and each **Cross-Join** operation is not enclosed between parentheses.

Since the “,” **Cross-Join** operator is associative, it does not make any difference if you add parentheses to enclose the individual **Cross-Join operations**. For this reason, the individual **Cross-Join operations** are not enclosed in parentheses. A nested **Cross-Join**

<sup>53</sup> You can find them in Query “F\_Join\_ON\_exp” in “Company\_Database.accdb”.

operation would therefore be usually written:

```
{ [[ ( {Table-name or Query-name} ) ] ] [AS Input-record-list-name_1]
or [ ( {Select-opr or Union-opr } ) ] AS Input-record-list-name_1 }
'
{ [[ ( {Table-name or Query-name} ) ] ] [AS Input-record-list-name_2]
or [ ( {Select-opr or Union-opr } ) ] AS Input-record-list-name_2 }
'
...
'
{ [[ ( {Table-name or Query-name} ) ] ] [AS Input-record-list-name_n]
or [ ( {Select-opr or Union-opr } ) ] AS Input-record-list-name_n }
```

Notice that **every input record-list** in the nested **Cross-Join** operation **must** have a **name**, either because it is a Table or Query, or because it is explicitly named with an “**AS**” clause.

The following is a concrete example of a **nested Cross-Join operation**<sup>54</sup>:

```
SELECT T_House_Owners.Address, Cars.Car, Non_capital_cities, Capital
FROM
  T_House_Owners
'
(
  SELECT Car
  FROM T_Car_Owners
  WHERE ID > 0
) AS Cars
'
F_Select_nested AS Q
'
(
  SELECT Capital
  FROM T_Capital_Cities
) AS Capitals
```

Finally, remind that a **Join** operation **cannot** be the **outermost** operation in a Query. If you have a **series** of nested **Join** operations, the outermost **Join** operation **must** be enclosed in a **Select** operation.

## F.9 What is a Union operation and how do I write it?

A **Union operation** is an SQL operation performed with a **Union** operator (click *F.9.1*) plus its corresponding operands. Therefore, a **Union operation** is the complete SQL code associated to the **Union** operator.

A simple example of a **Union operation** is:

```
SELECT City
FROM T_Cities_in_US
UNION ALL
SELECT City
FROM T_Cities_in_EU
```

The **output** record-list of the above **Union operation** is all the cities in the “**T\_Cities\_in\_US**” plus all the cities in the Table “**T\_Cities\_in\_EU**”.

A **Union** operation **can** be the **outermost** operation in a Query. However, in MS-Access my advice is that you use a **Select** operation to enclose the outermost **Union** operation

<sup>54</sup> This is the Query “**F\_Join\_Cross\_nested**” from the “**Company\_Database.accdb**” file.

**in the Query.** One reason for this is that MS-Access prevents you from configuring field formatting in a Query with an outermost **Union** operation (click *K.4.7*).

If you want to know more about a **Union operation**, you may click:

- “*F.9.1 What are the Union operators?*”
- “*F.9.2 What are the input record-lists of a Union?*”
- “*F.9.3 What are the output fields of a Union?*”
- “*F.9.4 What is the output record-list of a Union?*”
- “*F.9.5 How do I write a correct (syntax) Union?*”
- “*F.9.6 Why do I find misleading the names of the Union operators?*”

### **F.9.1 What are the Union operators?**

The **Union** operators are **consulting** (click *F.6*) operators that work over **two input** record-lists.

A simplified view of writing (syntax) a **Union operation** is:

```

Left-Input-record-list
Union-Operator
Right-Input-record-list

```

There are two **Union** operators: “**UNION ALL**” and “**UNION**”.

The “**UNION ALL**” operator produces an **output** record-list which is the **mixing** of its **two input** record-lists. This is, the **output** record-list contains **all the records** from its **two input** record-lists in an **unknown** order.

The “**UNION**” operator produces an **output** record-list in which its **two input** record-lists are **mixed**, and **only one** record is produced from **each set** of duplicate records. Therefore, in the **output** record-list of a “**UNION**” operator, you will **never** find any duplicate records. Recall that for the purpose of discarding duplicate records **Nulls** are considered the same value. Notice that “**UNION**” is the same as enclosing a “**UNION ALL**” into a “**SELECT DISTINCT \***”.

For both “**UNION ALL**” and “**UNION**” it is **mandatory** that the **two input** record-lists have the **same number of fields**. If the **two input** record-lists do not have the same number of fields, the **Union operation** will **crash**.

As you may see, the only functional difference between both operators is that “**UNION ALL**” preserves all duplicate records, while “**UNION**” removes all redundant duplicate records. Notice that “**UNION**” is **slower** than “**UNION ALL**” because the system has to check for duplicate records, and remove the redundant ones, before producing the **output** record-list.

For the case in which its **two input** record-lists have the **same field names**, “**UNION ALL**” and “**UNION**” operators are commutative. This is, if you exchange its two **input** record-lists (that have the same field names), you get **exactly the same output** record-list.

The “**UNION ALL**” operator is **associative**, and the “**UNION**” operator is also

associative<sup>55</sup>. However, the combination of “UNION ALL” and “UNION” is **not associative**.

Notice that the **Union** operators do something **different** from the conventional **union** of sets. I therefore consider that the **name** of **Union** operators is somehow misleading (click [F.9.6](#)).

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click [“F.11.2 What are the SQL color codes used in this Guide?”](#).

## F.9.2 What are the **input record-lists** of a **Union**?

The **two input** record-list of the **Union** operators are written at **both** sides of the **Union** operator. One of the **input** record-list is written just before the **Union** operator, and the other one is written just after it, as follows:

```
[ ( { Union-opr or Select-opr } [ ] ]
{ UNION ALL or UNION }
[ ( { Union-opr or Select-opr } [ ] ]
```

Each of the **two input** record-lists can be written as **either** a **Select operation** or a **Union operation**. The **input** record-lists **cannot** be a **Table** name, a **Query** name nor a **Join operation**.

Notice that the two **input** record-lists of a **Union** operator **may be the same record-list**.

The **Select operation** or **Union operation** **may** be enclosed between parentheses and **cannot** have an “**AS**” clause. This writing rule is **different** in the **Select/Transform** and **Join** operations.

It is **mandatory** that **both input** record-lists have the **same number of fields**. Otherwise, the Query will **crash**.

However, the **data/field types** of the field pairs, field to field, in its **two input** record-lists **can be different**. This may surprise you, because it seems odd that for example the field type of field “**i**” in one **input** record-list is **Number-Long**, while the data/field type of field “**i**” in the **other input** record-list is a **Date** or **Date/Time** and yet the **Union operation** will produce a valid **output** record-list.

In fact, a valid **output** record-list is produced on **most** cases because MS-Access performs a conversion of the data types, producing a correct **output** data/field type, and correct **output** field **values**, for each of the **input** field **pairs** that have **different** data types. If you want to know more about this, you may click [“F.9.3 What are the output fields of a Union?”](#).

## F.9.3 What are the **output fields** of a **Union**?

The **number of output fields** of the **Union** operators is the same as the **number of input fields** of either of its **input** record-lists.

### **What are the output field names of a Union?**

The **output field names** of the **Union** operators are the same as the **input field names** of its **first (left) input** record-list. The names of the **second (right) input** record-list are

---

<sup>55</sup> There could be some type conversion combinations that make Union operators non-associative.

ignored.

### What is the output field order of a Union?

The **output field order** of the **Union** operators is the same as the **input field order** of its **first (left) input** record-list.

### What are the output data/field types of a Union?

If the **two input fields** in the same order position “i” in both **input** record-lists have the **same data/field type**, then **that same data/field type** will be the one of the corresponding **output** field in the same order position “i”.

However, if the **two input fields** “i” have a **different data/field types**, then the **data/field type** of the corresponding **output** field “i” will be **such** that it can represent **the range of values** of **both input** data/field types with the **highest precision possible**.

To clarify this, I will now present a listing of concrete cases of **input** data/field type pairs, and its resulting **output** data/field type. Remind that in this Lightning Guide “**integer-like**” data/field types are *Boolean*, *Byte*, *Integer*, *Long* and *LongLong*. The listing of concrete cases is:

- **Union** of any combination of two **integer-like** data/field types:  
Result is the **integer-like data type**, among the two, with a **larger** range of values. Remind that a *Boolean* is stored in two bytes, this is, for this purpose it is equivalent to an *Integer*.
- **Union** of *Currency* with any **integer-like data type** (except *LongLong*):  
Result is *Currency*.
- **Union** of *LongLong* with *Currency*:  
Result is *Double*.
- **Union** of any **integer-like data type** with *Single*:  
Result is *Single*.
- **Union** of any **integer-like data type** with *Double*:  
Result is *Double*.
- **Union** of *Currency* with *Single*:  
Result is *Double*.
- **Union** of *Double* with any **integer-like data type**, *Currency* or *Single*:  
Result is *Double*.
- **Union** of *String* or *Date* with any **other data type**:  
Result is *String*.  
*Date* values are represented as a text string (e.g., the **string** “4/5/2003”). Numeric values are represented as a text string (e.g., the **strings** “145” or “34.47”).

In all the cases above, when converting a *Boolean* value to a **number** or a **string**, **True/Yes/On** or **ticked** is represented as the number “-1” or the text string “-1”, respectively, and **False/No/Off** or **unticked** is represented as the number “0” or the text string “0”, respectively.

### What are the output field values and/or the output record-list of a Union?

If you want to know what are the **output field values** and/or what is the **output record-**



list of a **Union**, you may click “[F.9.4 What is the output record-list of a Union?](#)”.

### **F.9.4 What is the output record-list of a Union?**

You may click:

- “[F.9.4.1 What are the output field values of a Union?](#)”
- “[F.9.4.2 What are the output records of a Union?](#)”
- “[F.9.4.3 How many output records does a Union produce?](#)”

If you rather want to know what are the **output fields** of **Union**, you may click “[F.9.3 What are the output fields of a Union?](#)”.

#### **F.9.4.1 What are the output field values of a Union?**

The field **values** in each **output** record are **exactly the same** as the ones of its corresponding **input** record, **unless** the data/field types of a given **input field pair** are **different**.

If the data/field types of a given **input field pair** are **different**, the **output** data/field type will be **different to one, or both, input** data/field types. This implies that **data type conversion** will be performed over the **values** of the **input** field(s) whose data/field type is **different** from its corresponding **output** field (click [F.9.3](#)).

When performing **data type conversion**, the field **values** on some **output** records **may be changed**. If you want to know more about changed values arising from data type conversion you may click “[G.2.5 How do I force a value to belong to a specific data type?](#)”.

#### **F.9.4.2 What are the output records of a Union?**

The **output** records of the “**UNION ALL**” operator are the **mixing** of the records from its **two input** record-lists. This is, the **output** record-list contains **all the records** from its **two input** record-lists in an **unknown** order.

The **output** records of the “**UNION**” operator are the **mixing** of the records from its **two input** record-lists, and **only one** record is produced from **each set** of duplicate records. Therefore, in the **output** record-list of a “**UNION**” operator, you will **never** find any duplicate records. Recall that for the purpose of discarding duplicate records **Nulls** are considered the same value.

#### **F.9.4.3 How many output records does a Union produce?**

Knowing how many records a **Union** produces is **very** useful when **debugging** your Queries.

A “**UNION ALL**” produces **num(L)+num(R)** records.

A “**UNION**” produces **num(L)+num(R)** **minus** the number of **redundant** duplicate records.

In the expressions above, “**L**” is the **left (first) input** record-list, and “**R**” is the **right (second) input** record-list.

The function “**num()**” represents the number of records in the record-list provided as its argument.



## F.9.5 How do I write a correct (syntax) Union?

You may click:

- “F.9.5.1 What is a syntax-example of a Union?”
- “F.9.5.2 What are the formal rules (syntax) to write a Union?”
- “F.9.5.3 How do I nest Unions?”

### F.9.5.1 What is a syntax-example of a Union?

An illustrative example of a fairly complete **Union operation**<sup>56</sup> is:

```
(
  (SELECT ID AS New_1, Name AS New_2, Address AS New_3
   FROM T_House_Owners )
  UNION
  SELECT ID, Name, Len(Car)
  FROM T_Car_Owners
  UNION
  SELECT Capital, District, Temp_Min
  FROM T_Capital_Temps
)
UNION ALL
(
  SELECT Capital, Cal_Year, Rainfall
  FROM T_Capital_Rainfall
  UNION ALL
  SELECT ID, Name, Len(Car)
  FROM T_Car_Owners
  UNION ALL
  SELECT Capital, District, Temp_Min
  FROM T_Capital_Temps
)
```

The first **Select** operation is enclosed between **parentheses**, but the other ones are not, because parentheses are optional.

Remind that the “**UNION**” operator is associative, and the “**UNION ALL**” is also associative (but their combination is **not** associative). For this reason, you can safely write a **series** of **Select** operations **connected** with either “**UNION**” or “**UNION ALL**” operators, as shown in this example.

Notice that **each** of the **two** series of “**UNION**” and “**UNION ALL**” operations in the example above are enclosed between parentheses, and both series are connected with an additional “**UNION ALL**” operator.

If you **added** or **removed one** field to/from **any** of the **Select** operations, the Query would **crash**, because the **number of fields must** be the same in the **two input** record-lists of the **Union** operators.

Notice that the **data/field type** of the fields placed **in the same position** is **not** the same. For the **1<sup>st</sup> field**, all data/field types are **Short Text** or **String**. However, for the **2<sup>nd</sup> field**, the **fourth Select** has field type **Number-Double** (field “**Cal\_Year**”) while all the other ones have field type **Short Text**. Also, for the **3<sup>rd</sup> field**, the **first Select** has field type **Short Text**, the **second** and **fifth Select** has data type **Double** and the remaining **Selects** have field type **Number-Long**. This is perfectly accepted in a **Union** operation, and type

<sup>56</sup> This is the Query “**F\_Union\_Syntax**” from the “**Company\_Database.accdb**” file.

conversion will be performed (click *F.9.3*).

Finally, notice that the **output field names** from this **Union** operation will be “**New\_1**”, “**New\_2**” and “**New\_3**”, because these are the field names of the **first** (i.e., leftmost) **Select** operation. See how I assigned these field names with the “**AS**” clause in the **first Select** operation.

### F.9.5.2 What are the formal rules (syntax) to write a Union?

A correct **Union operation** has to be written in the following way:

```
[ ( ) { Union-opr-A or Select-opr-A } [ ] ]
{ UNION ALL or UNION }
[ ( ) { Union-opr-B or Select-opr-B } [ ] ]
```

The words in **bold** font are SQL **keywords**. The parentheses enclosed in square brackets “[ ]” are optional. The elements separated with “or” are alternative options. Each list of alternative options for an element is enclosed between curly braces “{ }” when the element is **not optional**. Some curly braces “{ }” are colored just to make it easier to see which ones are paired.

All the terms above with a trailing “-opr” are SQL operations.

It is **mandatory** that **both input** records-lists have the **same number of fields**. Otherwise, the Query will **crash** with a syntax error message.

Notice that the two **input** record-lists of a **Union** operator **may be the same record-list**.

The outermost series of **Union** operations of a Query may be enclosed between parentheses.

### F.9.5.3 How do I nest Unions?

The **Union operations** can be nested, and you will **very frequently** write a **Union operation** as a series of **Select operations** connected by means of “**UNION**” or “**UNION ALL**” operators. Neither the interior **Union operations**, nor each of the **Select operations**, have to be enclosed in parentheses. This means that the following **Union operation** is correct:

```
Select-opr-1
UNION ALL
Select-opr-2
UNION ALL
...
UNION ALL
Select-opr-n
```

The **output** record-list from the above **Union operation** would be **all the records** produced by all the **Select operations 1** through **n**. The field names of the **output** record-list are the ones of the **first (leftmost) input record-list** (i.e., the field names of “**Select-opr-1**”). Therefore, the **field names** of the second and subsequent **Select operations** are **completely ignored**.

You may combine both “**UNION**” or “**UNION ALL**” operators with a series of **Select operations** in a row without parentheses. This means that the following **Union operation**

is correct.

```

Select-opr-1
UNION ALL
Select-opr-2
UNION
Select-opr-3
UNION ALL
...
UNION
Select-opr-n

```

However, in spite of this being a **correct** SQL piece of code, I strongly **recommend** avoiding it. You may write a series of “**UNION**” operators in a row, or a series of “**UNION ALL**” operators in a row, and this is not a problem because the “**UNION**” operator is associative, and also the “**UNION ALL**” operator is associative<sup>57</sup>. However, the combination of “**UNION**” and “**UNION ALL**” operators **is not associative**. Therefore, if you combine “**UNION**” and “**UNION ALL**” **without** parentheses, you will very likely make a **mistake** in your intended result, because the **output** record-list depends on the evaluation order rules of the expression. If you combine “**UNION**” and “**UNION ALL**” operators in a row, I strongly recommend that you add parentheses, to indicate explicitly the evaluation order that you want.

### F.9.6 Why do I find misleading the names of the Union operators?

I find misleading the names of the **Union** operators for two reasons.

On the one hand, the conventional mathematical union “U” operation defined between **sets** produces **all** the elements from **both** sets, except **the same** elements in both sets, that are **not** replicated in the result.

Neither “**UNION ALL**” nor “**UNION**” do this. Operator “**UNION ALL**” produces **all** the records from its **both** operands, regardless of some records being **the same** in both **input** record-lists. Operator “**UNION**” mixes **all** the records from its **both** operands, and then removes **all** redundant duplicates. Therefore, “**UNION**” removes **all** redundant duplicates **between both input** record-lists, but it **also** removes **all** redundant duplicates from **each** of the **input** record-lists, even if they were not **among** its two **input** record-lists. I am aware that in set theory, each of the sets cannot contain duplicate elements, but this only reinforces the fact that the union operator in set theory is different from the **Union** operators in SQL.

On the other hand, the “**UNION**” name contains nothing to remind/highlight that it **removes duplicates**. It therefore happens now and then to mistakenly use “**UNION**” instead of “**UNION ALL**”.

I think that naming them “**MIX**” and “**MIX DISTINCT**” instead of “**UNION ALL**” and “**UNION**”, respectively, would have been a better choice.

### F.10 What is a Transform operation and how do I write it?

This chapter also answers the questions:

- **How do I create cross tables?**

---

<sup>57</sup> There could be some type conversion combinations that make **Union** operators non-associative.

- **What is a crosstab Query?**

A **Transform operation** is an SQL operation performed with the **Transform** operator (click *F.10.1*) plus its corresponding operands. Therefore, a **Transform** operation is the complete SQL code associated to the **Transform** operator.

A **Transform** operation produces cross tables in a very flexible way. In its **simplest** form, a **Transform** operation **generates new field names** from the **values** of the **field** written in the “**PIVOT**” clause, **and also**, places the **values** of the **expression** written in the “**TRANSFORM**” clause in the **correct** places **under** the **newly generated field names**.

**Transform** is **extremely** useful to **display** your Query **results** as a **cross table**. Imagine you have the following record-list (in this case it is a Table<sup>58</sup>, but most frequently it is a Query result or an inner SQL operation):

T_Capital_Rainfall		
Capital	Cal_Year	Rainfall
Beijing	2018	25
Beijing	2019	41
Washington	2018	22
Washington	2019	17
Beijing	2020	27
Beijing	2021	38
Washington	2020	26
Washington	2021	9

Now you want to display this information as a **cross table**, with **fields** “**Capital**” (renamed to “**Cap\_City**”), “**2018**”, “**2019**”, “**2020**” and “**2021**”, and the **value** of “**Rainfall**” in the **corresponding** cell **under** the **fields** “**2018**”, “**2019**”, “**2020**” and “**2021**”. In summary, what you want is:

F_Transform_1				
Cap_City	2018	2019	2020	2021
Beijing	25	41	27	38
Washington	22	17	26	9

Then, this is **exactly** what you would get with the following **Transform** operation<sup>59</sup>:

```
TRANSFORM First(Rainfall) AS GenVals
SELECT Capital AS Cap_City
FROM T_Capital_Rainfall
GROUP BY Capital
PIVOT Cal_Year ;
```

The “**n**” **leftmost** output fields of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are

<sup>58</sup> This is the Table “**T\_Capital\_Rainfall**” in file “**Company\_Database.accdb**”.

<sup>59</sup> This is the Query “**F\_Transform\_1**” from the “**Company\_Database.accdb**” file.

**generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click *F.10.11*), subject to the modifications of the optional “**IN**” **list** (click *F.10.12*). In this example, we have one “**SELECT**” **field** (field name “**Cap\_City**”) and four “**PIVOT**” **fields** (with field **names** “**2018**”, “**2019**”, “**2020**” and “**2021**”).

**Transform** operations **cannot** be nested. The **Transform** operation must be **the first** operation in a Query, and a Query **cannot contain** any other **Transform** operation. Consequently, a **Transform** operation **cannot** be used as an **input record-list** in any other Query or SQL operation. A Query that has a **Transform** operation **cannot** be used in any other Query. The **Transform** operation can therefore only be the **very last** operation over a record-list.

If you want to know more about a **Transform** operation, you may click:

- “*F.10.1 What is the Transform operator?*”
- “*F.10.2 What is the input record-list ( “FROM” clause) of a Transform?*”
- “*F.10.3 What are the output fields of a Transform?*”
- “*F.10.4 What is the output record-list of a Transform?*”
- “*F.10.5 Can I see an example of a Transform operation?*”
- “*F.10.6 What is the “TRANSFORM” clause of a Transform?*”
- “*F.10.7 What is the “SELECT” clause of a Transform?*”
- “*F.10.8 What is the “ORDER BY” clause of a Transform?*”
- “*F.10.9 What is the “WHERE” clause of a Transform?*”
- “*F.10.10 What is the “GROUP BY” clause of a Transform?*”
- “*F.10.11 What is the “PIVOT” clause of a Transform?*”
- “*F.10.12 What is the “IN” clause of a Transform?*”
- “*F.10.13 How do the clauses from Transform and Select compare?*”
- “*F.10.14 How do I write a correct (syntax) Transform?*”

### **F.10.1 What is the Transform operator?**

**Transform** is a **consulting** (click *F.6*) operator that works over one single **input record-list**, plus other operands that are not record-lists.

The **Transform** operator includes the clauses “**SELECT**”, “**DISTINCT**” (optional and irrelevant), “**DISTINCTROW**” (optional and not advisable), “**FROM**”, “**WHERE**” (optional), “**GROUP BY**”, “**ORDER BY**” (optional) and “**IN**” (optional).

A simplified view of writing (syntax) a **Transform operation** is:

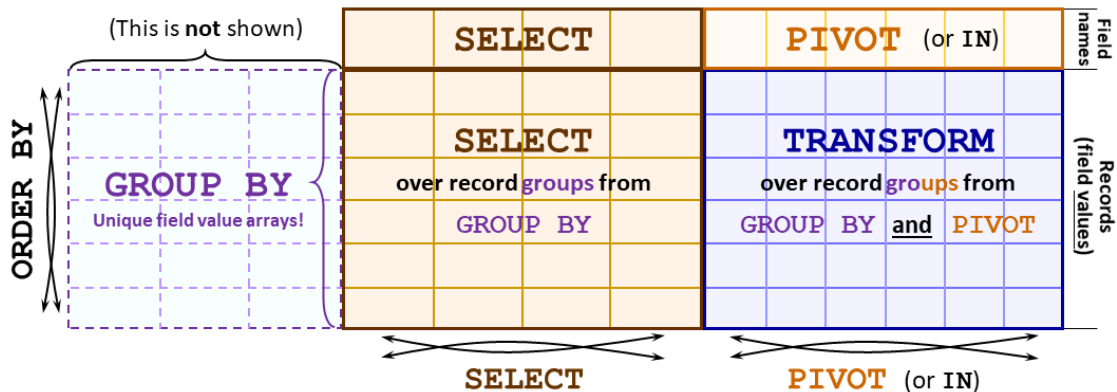
```

TRANSFORM PIVOT-field-values-expression()
SELECT [DISTINCT] [DISTINCTROW] Output-expression() 1 to n
FROM Input-record-list
[ WHERE Where-Boolean-expression(Input-field-names) ]
GROUP BY Group_by-expression(Input-field-names) 1 to k
[ ORDER BY Order_by-expression() 1 to w ]
PIVOT PIVOT-field-names-expression(Input-field-names) ;
[ IN ( List-of-PIVOT-values ) ]

```

The clauses enclosed between square brackets “[ ]” are optional.

The following picture shows the **output** of the **Transform operation**:



The **Transform** operator creates a **cross table** by converting the **distinct values** of the “**PIVOT**” **expression** (click [F.10.11](#)) into **additional generated field names** (i.e., “**columns**”) and presenting the **results** of the “**TRANSFORM**” **expression** (click [F.10.6](#)) in the corresponding **cells** (**row x column**).

The **Transform** operator produces:

- The **rows** by doing record **aggregation** over the (mandatory) “**GROUP BY**” **expressions**.
- The **fixed** leftmost **columns** (the “**SELECT**” **fields names** and **values**) with the conventional “**SELECT**” **expressions**.
- The **cross table column names** (the “**PIVOT**” **field names**) by doing record **aggregation** over the (mandatory) “**PIVOT**” **expression**,
- The **cross table column values** by doing record **aggregation** of the “**TRANSFORM**” **expression** over the “**GROUP BY**” **expressions** jointly with the “**PIVOT**” **expression**.

Remind that record aggregation produces **only one value** from **each group** of records. I now explain the above bullet points in more detail.

The **number** of **rows** is determined by the **1 to k** “**GROUP BY**” **expressions** (click [F.10.10](#)). There will be **as many rows** as **groups** of (retained) **input records** (see below). **Each group** contains the (retained) **input records** that produce the **same results** in all the **1 to k** “**GROUP BY**” **expressions**. Notice that the resulting **arrays** of “**k**” **results** from the “**GROUP BY**” **expressions** are **all distinct**. Notice also that these **result arrays** are **not shown** in the **output** of the **Transform operation** and are just internal for its processing. The “**GROUP BY**” clause of a **Transform** works exactly the



same as the “**GROUP BY**” clause (click *F.7.9*) of a **Select**.

The “**n**” **leftmost output fields** of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are **generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click *F.10.11*), subject to the modifications of the optional “**IN**” **list** (click *F.10.12*).

The number of **select columns** “**n**”, their **field names**, the **field order** and the **field values** are determined by the “**SELECT**” clause (click *F.10.7*) with the “**GROUP BY**” clause (click *F.10.10*). The **value** of field “**j**” from the **row** “**i**” is produced by applying the corresponding “**SELECT**” **expression** “**j**”, to **all** the records in the **group** “**i**” of (retained) **input records** corresponding to **this row**. These **groups** of (retained) **input records** are **the same** as the ones indicated in the previous paragraph.

The **number** of “**PIVOT**” **columns** and their **field names** are determined by the **distinct values** returned by the “**PIVOT**” **expression** computed over **all** the (retained) **input records**. The **field order** is (click *F.10.3.3*), left to right, **ascending alphanumeric** by the **value** returned by the “**PIVOT**” **expression**. However, if the optional “**IN**” clause is used, then the **number** of “**PIVOT**” **columns**, the **field names** and the **field order** are **all** determined by the “**IN**” **list** (click *F.10.12*).

The **field value** of “**PIVOT**” **field** “**j**” from the **row** “**i**” is produced by applying the “**TRANSFORM**” **expression**, to **all** the records in the **group** “**(i, j)**” of (retained) **input records** corresponding to **row** “**i**” and **column** “**j**”. Each **group** “**(i, j)**” contains the (retained) **input records** that produce the **same results** in **all** the **1** to **k** “**GROUP BY**” **expressions** as the **values** in **row** “**i**” **and** the **same result** in the “**PIVOT**” **expression** as the **field name** of column “**j**”. If there is a **field name collision** with a “**SELECT**” **field**, or the “**IN**” clause is used, this becomes slightly more complex (click *F.10.4.1*).

The order of records is **unknown**, unless the optional “**ORDER BY**” clause (click *F.10.9*) is used. The ordering works the same as the “**ORDER BY**” clause (click *F.7.12*) from the **Select** operator, with just one restriction: the “**ORDER BY**” **expressions** **must** be a **list** of **any number** of **exactly the same** “**GROUP BY**” **expressions** or the “**PIVOT**” **expression**.

The “**Input-record-list**” is stated in the “**FROM**” clause. The “**FROM**” clause in the **Transform** operator works exactly the same as the “**FROM**” clause (click *F.7.4*) in the **Select** operators.

If the optional “**WHERE**” clause is used, the **input records** that produce *True* in the “**Where-Boolean-expression()**” are **retained**, while the other ones are **discarded**. The “**WHERE**” clause (click *F.10.9*) of a **Transform** operator works exactly the same as the “**WHERE**” clause (click *F.7.7*) of a **Select** operator. If against my advice you use the optional “**DISTINCTROW**” clause, you may click *F.7.8*.

Very important to highlight that if the **input record-list** is a Table name, no records will be **deleted** from the Table, and the Table remains **unmodified**. This is so because all the SQL **consulting** operators work over an **image** of Table’s records, and **not** over the Table records themselves. Remind that the **Transform** operator is a **consulting** SQL operator (i.e., one that can only consult Tables) and it is not a **data-changing** SQL operator. If you want to actually **modify** your Table’s records, you may click



[“F.6.2 What are the SQL data-changing operators?”](#).

If you want to know more about **Transform**, you may click:

- [“F.10.2 What is the input record-list \(“FROM” clause\) of a Transform?”](#)
- [“F.10.3 What are the output fields of a Transform?”](#)
- [“F.10.4 What is the output record-list of a Transform?”](#)
- [“F.10.5 Can I see an example of a Transform operation?”](#)
- [“F.10.13 How do the clauses from Transform and Select compare?”](#)
- [“F.10.14 How do I write a correct \(syntax\) Transform?”](#)

If you want to know the SQL color codes used in this **Lightning Guide**, you may click [“F.11.2 What are the SQL color codes used in this Guide?”](#).

## F.10.2 What is the input record-list (“**FROM**” clause) of a Transform?

In a **Transform** operation (click [F.10.1](#)), the mandatory “**FROM**” clause indicates its **input record-list**, as follows:

```
FROM {      [( {Table-name or Query-name} )]
      or [   {Table-name or Query-name}   [AS Input-record-list-name] ]
      or [ ( {Select-opr or Union-opr } ) [AS Input-record-list-name] ]
      or [( { Inner-or-Outer-Join-opr   } ) or Cross-Join-opr           ] }
```

The **input record-list** is indicated in its “**FROM**” clause, **exactly the same** as the one from the **Select** operation.

If you want to write (syntax) a correct “**FROM**” clause, you may click [F.10.14](#).

## F.10.3 What are the output fields of a Transform?

You may click:

- [“F.10.3.1 What is the number of output fields of a Transform?”](#)
- [“F.10.3.2 What are the output field names of a Transform?”](#)
- [“F.10.3.3 What is the output field order of a Transform?”](#)
- [“F.10.3.4 What are the output data/field types of a Transform?”](#)
- [“F.10.4 What is the output record-list of a Transform?”](#)

### F.10.3.1 What is the number of output fields of a Transform?

In a **Transform** operation (click [F.10.1](#)), the **number of output fields** is indicated in the “**SELECT**”, “**PIVOT**” and “**IN**” clauses as follows:

```
TRANSFORM PIVOT-field-values-exp()
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
      {
        * or
        Output-exp_1(exp-elements) [AS Output-field-name_1]
      [ , ...
        , Output-exp_n(exp-elements) [AS Output-field-name_n] ] }
...
PIVOT PIVOT-field-names-exp(Input-field-names)
[IN ( List-of-PIVOT-values ) ]
```

The “**n**” **leftmost output fields** of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are **generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click *F.10.11*), subject to the modifications of the optional “**IN**” **list** (click *F.10.12*).

**What is the number of output “**SELECT**” fields of a Transform?**

The **number “n”** of “**SELECT**” **fields** is the **number** of “**SELECT**” **expressions**. This is **exactly the same** as in a **Select** operation (click *F.7.5*).

**What is the number of output “**PIVOT**” fields of a Transform?**

If the optional “**IN**” clause is **not** used, the **number** of “**PIVOT**” **fields** is the **number** of **distinct values** returned by the “**PIVOT**” **expression** computed over **all** the (retained) **input records**.

Else, if the optional “**IN**” clause is **used**, then the **number** of “**PIVOT**” **fields** is the **number** of **values** in the “**IN**” **list**.

### F.10.3.2 What are the output field names of a Transform?

In a **Transform** operation (click *F.10.1*), the **output field names** are indicated in the “**SELECT**”, “**PIVOT**” and “**IN**” clauses as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
  {
    * or
    Output-exp_1(exp-elements) [AS Output-field-name_1]
  [ , ...
    , Output-exp_n(exp-elements) [AS Output-field-name_n] ] }
...
PIVOT PIVOT-field-names-exp (Input-field-names)
[IN ( List-of-PIVOT-values ) ]
```

The “**n**” **leftmost output fields** of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are **generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click *F.10.11*), subject to the modifications of the optional “**IN**” **list** (click *F.10.12*).

**What are the names of output “**SELECT**” fields of a Transform?**

The **names** of the “**n**” “**SELECT**” **fields** are the identifiers “**Output-field-name\_i**” from the “**SELECT**” clause. This is **exactly the same** as in a **Select** operation (click *F.7.5*).

**What are the names of output “**PIVOT**” fields of a Transform?**

Regarding the **names** of the “**PIVOT**” **fields**, we have the following three cases:

- The “**IN**” clause is **not** used, and the **distinct value** (converted to *String*) returned by the “**PIVOT**” **expression** is **different** from **all** the “**SELECT**” **field names**. Then, the **name** of this “**PIVOT**” **field** is the **distinct value** (converted to *String*) returned by the “**PIVOT**” **expression**.
- The “**IN**” clause is **used**, and the **value** (converted to *String*) from the “**IN**” **list** is

different from all the “**SELECT**” field names.

Then, the **name** of this “**PIVOT**” field is the **value** (converted to *String*) from “**IN**” list.

- The **value** (converted to *String*), which is either a **distinct value** returned by the “**PIVOT**” expression or a **value** from the “**IN**” list, is the **same as** one of the “**SELECT**” field names.

Then, the **name** of this “**PIVOT**” field is “**FieldN**”, where “**N**” is an integer value assigned by MS-Access.

The data type of the “**PIVOT**” expression is usually *String*, but it can also be any other data type. In case the data type of the expression is not *String*, then MS-Access will convert its result to a *String* (to become a field name). The values **True/Yes/On** or **ticked** and **False/No/Off** or **unticked** are converted to the field names “-1” and “0”, respectively. An integer-like data type value is converted to the field name of the equivalent *String* (e.g., “-12”, “14”). A *Date* value is converted to the field name of the equivalent *String* (e.g., “04/05/2003”). A fractional data type is converted to the field name of the equivalent *String*, but, replacing the period “.” with underscore “\_”. The reason for this is that names cannot contain a period “.” (click *D.2.5*). Therefore, the number “-0.45” is converted to the field name “-0\_45”.

In case that one or more **input records** make the expression in the “**PIVOT**” clause produce **Null**, these **Nulls** will produce a valid “**PIVOT**” field name which is “<>”. I strongly recommend that you **avoid** using the string “<>” in the “**List-of-PIVOT-values**” of the “**IN**” clause, to avoid confusion with a “**PIVOT**” field arising from a **Null** (which is most likely not intentional).

A **Null**, either returned by the “**PIVOT**” expression or written in the “**IN**” list, will produce “<>” as the “**PIVOT**” field name. If both a **Null** and an explicit “<>” happen, the field corresponding to **Null** will be named “<>”, while the field corresponding to “<>” will be named “**FieldN**”, where “**N**” is an integer value assigned by MS-Access.

Field names cannot include the period “.” character (click *D.2.5*). For this reason, every period “.” character in a **value** from the “**PIVOT**” expression or in a **value** from the “**IN**” list, will be converted to the underscore “\_” character in the resulting “**PIVOT**” field name. For example, the returned values numeric “3.4” and string “Oh.No” from the “**PIVOT**” expression produce the “**PIVOT**” field names “3\_4” and “Oh\_No” respectively; likewise, the values “54.6” and “Hi.there” found in the “**IN**” list produce the “**PIVOT**” field name “54\_6” and “Hi.there”, respectively.

If you use the “**IN**” clause, the **data type** returned by the “**PIVOT**” expression must be the same as the one of all the elements in the “**IN**” list, either **directly** or through **type conversion**. Otherwise, the Query will **crash**. Curiously, a **Null**, either **returned** by the “**PIVOT**” expression or explicitly **written** in “**IN**” list, is always considered as having a compatible data type. I now present a few examples to clarify this paragraph:

- The following works, because “**Cal\_Year**” is a **number** and **all** the elements in the “**IN**” list are **numbers**, or **can be converted** to a **number**, or are **Null**.

```
PIVOT Cal_Year
IN (2016, 2020, "2018", "34", Null)
```

- The following **crashes**, because “**Cal\_Year**” is a **number** and the **string** “**Hello**” cannot be converted to a **number**.

```
PIVOT Cal_Year
IN (2016, 2020, "2018", "Hello", Null)
```

- The following works, because “**Date\_Time**” is a *Date/Time* and **all** the elements in the “**IN**” **list** are either *Date/Time*, or can be converted to a *Date/Time*, or are **Null**.

```
PIVOT Date_Time
IN (#2018-1-1#, "3/January/2020", Null)
```

- The following **crashes**, because “**Date\_Time**” is a *Date/Time* and the **string** “**Hello**” cannot be converted to a **number**.

```
PIVOT Date_Time
IN (#2018-1-1#, "Hello", Null)
```

### F.10.3.3 What is the output field order of a Transform?

In a **Transform** operation (click *F.10.1*), the **output field order** is indicated in the “**SELECT**”, “**PIVOT**” and “**IN**” clauses as follows:

```
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
{
  * or
  Output-exp_1(exp-elements) [AS Output-field-name_1]
  [, ...
  , Output-exp_n(exp-elements) [AS Output-field-name_n] ] }
...
PIVOT PIVOT-field-names-exp(Input-field-names)
[IN ( List-of-PIVOT-values ) ]
```

The “**n**” **leftmost output fields** of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are **generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click *F.10.11*), subject to the modifications of the optional “**IN**” **list** (click *F.10.12*).

#### What is the order of output “SELECT” fields of a Transform?

The **order** of the “**n**” “**SELECT**” **fields** is the same as the one of the “**SELECT**” **expressions**. This is **exactly the same** as in a **Select** operation (click *F.7.5*).

#### What is the order of output “PIVOT” fields of a Transform?

If the optional “**IN**” clause is **not** used, the **order** of “**PIVOT**” **fields** is, left to right, **ascending** by the **value** produced by the “**PIVOT**” **expression**. Notice that if the “**PIVOT**” **expression** produces a **numeric value**, the **field** ordering will be **ascending numeric**, while if it produces a *String value* the field ordering will be **ascending alphabetical**, which is **different**. You can easily change between both by enclosing the “**PIVOT**” **expression** in a type conversion function (click *G.2.5*). If a **Null** is produced, it will **always** be the **first** (leftmost) “**PIVOT**” **field**, with **field name** “<>”.

Else, if the optional “**IN**” clause **is** used, then the “**PIVOT**” **field order** is the one you wrote in the “**IN**” **list** (click *F.10.12*).

Notice that the field order just described is **maintained** even when a “**PIVOT**” **field name** (either converted from the “**PIVOT**” **expression** or from the “**IN**” **list**) is the

same as a “**SELECT**” **field name**. In this case, the “**PIVOT**” **field name** is changed to “**FieldN**”, but its **position** within the “**PIVOT**” **fields** is **not** changed.

The “**PIVOT**” **field name** corresponding to a **Null** value (i.e., the field name “<>”) can **also** be reordered by including it in the “**IN**” **list**. I strongly recommend that you **never** use the string “<>” neither as a “**SELECT**” **field name**, nor as a **name** in the “**IN**” list, nor as a name produced by the “**PIVOT**” **expression**. If you use “<>” in either of these three cases, you create the risk of **mistaking** that field with a “**PIVOT**” **field** arising from **Null** (which is most likely not intentional).

### F.10.3.4 What are the output data/field types of a Transform?

In a **Transform** operation (click *F.10.1*), the **output data/field types** are indicated in the “**TRANSFORM**”, “**SELECT**”, “**PIVOT**” and “**IN**” clauses as follows:

```

TRANSFORM PIVOT-field-values-exp()
SELECT [DISTINCT] [TOP int [PERCENT]] [DISTINCTROW or ALL]
    { * or
      Output-exp_1(exp-elements) [AS Output-field-name_1]
    [, ...
      , Output-exp_n(exp-elements) [AS Output-field-name_n] ] }
    ...
PIVOT PIVOT-field-names-exp(Input-field-names)
[IN ( List-of-PIVOT-values ) ]

```

The “**n**” **leftmost** output fields of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are **generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click *F.10.11*), subject to the modifications of the optional “**IN**” **list** (click *F.10.12*).

#### What is the data/field type of output “**SELECT**” fields of a Transform?

The **data/field type** of the “**n**” “**SELECT**” **fields** are the ones of the corresponding “**SELECT**” **expressions**. This is **exactly the same** as in a **Select** operation (click *F.7.5*).

#### What is the data/field type of output “**PIVOT**” fields of a Transform?

Regarding the data/field **type** of the “**PIVOT**” **fields**, we have the following three cases:

- The “**IN**” clause is **not** used, and the **distinct value** (converted to *String*) returned by the “**PIVOT**” **expression** is **different** from **all** the “**SELECT**” **field names**. Then, the **data type** of this “**PIVOT**” **field** is the one of the “**TRANSFORM**” **expression** “**PIVOT-field-values-exp()**”.
- The “**IN**” clause is **used**, and the **value** (converted to *String*) from the “**IN**” **list** is **different** from **all** the “**SELECT**” **field names**. Then, the **data type** of this “**PIVOT**” **field** is **unknown** (and all this field’s values are **Null**).
- The **value** (converted to *String*), which is either a **distinct value** returned by the “**PIVOT**” **expression** or a **value** from the “**IN**” **list**, is the **same** as one of the “**SELECT**” **field names**. Then, the **data type** of this “**PIVOT**” **field** is the one of the **same-name** “**SELECT**” **field**.

If you want to know what are the **output field values** and/or what is the **output record-list** of a **Transform**, you may click “[F.10.4 What is the output record-list of a Transform?](#)”.

### F.10.4 What is the output record-list of a Transform?

You may click:

- “[F.10.4.1 What are the output field values of a Transform?](#)”
- “[F.10.4.2 What are the output records of a Transform?](#)”
- “[F.10.4.3 How many output records does a Transform produce?](#)”

#### F.10.4.1 What are the output field values of a Transform?

In a **Transform**, the **output field values** are determined by the “**TRANSFORM**”, “**SELECT**”, “**PIVOT**” and “**IN**” clauses as follows:

```

TRANSFORM
  PIVOT-field-values-exp      ( Output-field-names
                                , PIVOT-field-names-exp(Input-field-names)
                                , Group_by-exp_1(Input-field-names)
                                , ...
                                , Group_by-exp_k(Input-field-names)
                                , SQL_agg_func(exp_t11(INOUT-field-names))
                                , ...
                                , SQL_agg_func(exp_t1d(INOUT-field-names))
                                ) [AS Output-values-Identifier]

SELECT [DISTINCT] [DISTINCTROW or ALL]
  Output-exp_1( Output-field-names
                , PIVOT-field-names-exp(Input-field-names)
                , Group_by-exp_1(Input-field-names)
                , ...
                , Group_by-exp_k(Input-field-names)
                , SQL_agg_func(exp_o11(INOUT-field-names))
                , ...
                , SQL_agg_func(exp_o1y(INOUT-field-names))
                ) [ AS Output-field-name_1 ]
  [ , ...
    , Output-exp_n( Output-field-names
                    , PIVOT-field-names-exp(Input-field-names)
                    , Group_by-exp_1(Input-field-names)
                    , ...
                    , Group_by-exp_k(Input-field-names)
                    , SQL_agg_func(exp_on1(INOUT-field-names))
                    , ...
                    , SQL_agg_func(exp_onz(INOUT-field-names))
                    ) [ AS Output-field-name_n ] ]
  ...
  PIVOT PIVOT-field-names-exp(Input-field-names)
[IN ( List-of-PIVOT-values ) ]

```

The “**n**” **leftmost output fields** of a **Transform** are called the “**SELECT**” **fields**, because they are determined by the “**SELECT**” clause. The **output fields** placed to the **right** of the “**SELECT**” **fields** are called the “**PIVOT**” **fields**, because they are **generated** by converting to *String* the **returned values** of the “**PIVOT**” **expression** (click [F.10.11](#)), subject to the modifications of the optional “**IN**” **list** (click [F.10.12](#)).

#### What are the values of output “SELECT” fields of a Transform?

The **values** of the “**n**” “**SELECT**” **fields** are the result of the “**SELECT**” **expressions**,



computed over the **groups** of (retained) **input records** produced by the “**GROUP BY**” **expressions**. This is the same as in a **Select-group\_by\_aggreg** operation (click *F.7.6.2*), with only one difference. The one difference is that in a **Transform**, the “**SELECT**” **expressions 1 to n** can also use as an **element** the “**PIVOT**” **expression**. You may see this in the SQL code above. This is somehow **surprising**, because the “**PIVOT**” **expression** produces **multiple** values (**different** in the general case) when computed over the (retained) **input records** in the **group** corresponding to a given **output** record (remind that we are using a “**GROUP BY**” clause). The way this works is that when evaluating a “**SELECT**” **expression** that includes the “**PIVOT**” **expression**, it returns the result **as if** the “**PIVOT**” **expression** was **enclosed** in the “**Min()**” SQL aggregate function. This is, whenever you write the the “**PIVOT**” **expression** “**PIVOT-field-names-expression()**” within a “**SELECT**” **expression**, it works **as if** you had written “**Min(PIVOT-field-names-expression())**”.

Notice that if a “**PIVOT**” **field** is discarded because the “**IN**” clause is used, and the corresponding **result** from the “**PIVOT**” **expression** is not in the “**IN**” **list**, the **result** of using the “**PIVOT**” **expression** within the “**SELECT**” **expression** will remain unaffected. This is, you will still get the “**Min()**” **value** over **all** the values produced, even if the “**Min()**” **value** is **not** in the “**IN**” **list**.

Finally, as a rather strange case, if you use the “**PIVOT**” **expression** as an **element** of a “**SELECT**” **expression**, and you also use the “**IN**” clause, the corresponding “**SELECT**” **expression** will produce the exception value “**#Error**”. Curiously, if the “**SELECT**” **expression** is **exactly the same** as the “**PIVOT**” **expression**, this works fine. This may be an MS-Access **bug**.

### What are the values of output “**PIVOT**” fields of a Transform?

Regarding the **values** of the “**PIVOT**” **fields**, we have the following three cases:

- The “**IN**” clause is **not** used, and the **distinct value** (converted to *String*) returned by the “**PIVOT**” **expression** is **different** from **all** the “**SELECT**” **field names**. Then, the **value** of this “**PIVOT**” **field** is the result of the “**TRANSFORM**” **expression** computed over the **groups** of (retained) **input records** produced by the “**GROUP BY**” **expressions** **jointly with** the “**PIVOT**” **expression**. In more detail, the **value** of **field “j”** and **row “i”** is produced by applying the “**TRANSFORM**” **expression**, to **all** the records in the **group** “**(i, j)**” of (retained) **input records** corresponding to **row “i”** and **column “j”**. Each **group** “**(i, j)**” contains the (retained) **input records** that produce the **same results** in **all** the **1 to k** “**GROUP BY**” **expressions** as the **values** in **row “i”** **and** the **same result** in the “**PIVOT**” **expression** as the **field name** of column “**j**”. Since these **groups** of **input records** depend on the result of the “**PIVOT**” **expression**, they will be different (in the general case) for **each** “**PIVOT**” **field**, and therefore, the “**TRANSFORM**” **expression** will produce **different values** (i.e., different **columns** of **values**) below **each** of the “**PIVOT**” **fields**.
- The “**IN**” clause is **used**, and the **value** (converted to *String*) from the “**IN**” **list** is **different** from **all** the “**SELECT**” **field names**. Then, the **value** of this “**PIVOT**” **field** is **Null** in **all** the **output** records.
- The **value** (converted to *String*), which is either a **distinct value** returned by the



“PIVOT” **expression** or a **value** from the “IN” **list**, is the same as one of the “SELECT” **field names**.

Then, the **value** of this “PIVOT” **field** is the same as the one in the same-name “SELECT” **field** in each and every output record. This is, the “column” of **values** under this **field** is the same as the “column” of **values** under the same-name “SELECT” **field**.

In the first bullet above, notice that it is **not surprising** to be able to use the “PIVOT” **expression** as one of the **elements** of the “TRANSFORM” **expression**, because in this case the “PIVOT” **expression** produces the same value in every record of each group. However, in the explanation above in “*What are the values of output “SELECT” fields of a Transform?*” it was surprising that you could use the “PIVOT” **expression** as an **element** of the “SELECT” **expressions**, because in that case the “PIVOT” **expression** may produce **different** values in the **different** records of each group.

Why does the same expression produce different values within the “SELECT” and “TRANSFORM” expressions?

Because in a Transform, the “SELECT” **expressions** and the “TRANSFORM” **expression** are computed over **different groups** of **input records**. The “SELECT” **expressions** are computed over **groups** of **input records** that produce the same results in **all** the “GROUP BY” **expressions**. However, the “TRANSFORM” **expression** is computed over **groups** of **input records** that produce the same result in **all** the “GROUP BY” **expressions** and also in the “PIVOT” **expression**.

Let me show this with an example over the Table “T\_Capital\_Rainfall\_District”.

T_Capital_Rainfall_District			
Capital	District	Cal_Year	Rainfall
Beijing	Dongcheng	2018	14
Beijing	Xicheng	2018	11
Beijing	Dongcheng	2019	18
Beijing	Xicheng	2019	23
Washington	Downtown	2018	10
Washington	Bloomington	2018	12
Washington	Downtown	2019	8
Washington	Bloomington	2019	9

If you now run the Query<sup>60</sup>:

```
TRANSFORM Sum(Rainfall) AS GenVals
SELECT Capital AS Cap_City, Sum(Rainfall) AS Total
FROM T_Capital_Rainfall_District
GROUP BY Capital
PIVOT Cal_Year ;
```

<sup>60</sup> This is the Query “F\_Transform\_sum” from the “Company\_Database.accdb” file.

you then get the result:

F_Transform_Sum			
Cap_City	Total	2018	2019
Beijing	32	14	18
Beijing	34	11	23
Washington	18	10	8
Washington	21	12	9

Notice how the “**TRANSFORM**” expression “**Sum(Rainfall)**” and the “**SELECT**” expression “**Sum(Rainfall)**”, that are exactly the same, produce **different** results under the field names “**Total**”, “**2018**” and “**2019**”. The reason is that the “**SELECT**” expression “**Sum(Rainfall)**” is computed over the record **groups** produced by the “**GROUP BY**” expressions, while the same “**TRANSFORM**” expression “**Sum(Rainfall)**” is computed over the record **groups** produced by the “**GROUP BY**” expressions jointly with the “**PIVOT**” expression. In other words, the “**SELECT**” expression “**Sum(Rainfall)**” is the **sum** for each “**Capital**” (i.e., adding **all districts and all years**) while the “**TRANSFORM**” expression “**Sum(Rainfall)**” is the **sum** for each “**Capital**” and each “**Cal\_Year**” (i.e., adding **all districts**). Notice how the “**TRANSFORM**” expression produces a **different column** of **values** for each “**PIVOT**” field (i.e., for “**2018**” and “**2019**”).

Since in this case we are using the “**Sum()**” aggregate function, the result under “**Total**” is the addition of the results under “**2018**” and “**2019**”. However, notice that this **does not** happen with some other SQL aggregate functions (see the last bullet point of *F.10.5*).

#### F.10.4.2 What are the output records of a Transform?

In a **Transform**, the **output records** are determined by the optional “**WHERE**” clause, the “**GROUP BY**” clause and the optional “**ORDER BY**” clause as follows:

```
[WHERE Where-Boolean-exp(Input-field-names) ]
GROUP BY      Group_by-exp_1(Input-field-names)
              [ , ...
              , Group_by-exp_k(Input-field-names) ]

[ORDER BY      Group_by-exp_x(Input-field-names) [DESC]
              [ , ...
              , Group_by-exp_y(Input-field-names) [DESC] ] ]
```

**Transform** produces **as many records** as **groups** of (retained) **input records** from the “**GROUP BY**” expressions. Each **group** contains the (retained) **input records** that produce the **same results** in **all** the “**GROUP BY**” expressions. Notice that the resulting **field value arrays** from “**Group\_by-expression()**” **1** to **k** are **all distinct**. Notice also that these resulting **field value arrays** are not shown in the result of the **Transform** operation.

For each such **output record** its field **values** will be the ones indicated in the previous subsection *F.10.4.1*.

The order of records is **unknown**, unless the optional “**ORDER BY**” clause

(click *F.10.12*) is used. The ordering works the same as the “**ORDER BY**” clause (click *F.7.12*) from the **Select** operator, with just one restriction: the expressions in the **Transform** “**ORDER BY**” clause **must** be a **list of any number of exactly the same “GROUP BY” expressions**.

### F.10.4.3 How many output records does a Transform produce?

Knowing how many records a **Transform** produces is **very** useful when **debugging** your Queries.

A **Transform** produces **as many records** as **groups** of (retained) **input records** are produced by the “**GROUP BY**” **expressions**. Each **group** contains the (retained) **input records** that produce the **same results** in all the “**GROUP BY**” **expressions**.

Notice that the **number of output records** is determined only by the (retained) **input records** (i.e., the “**WHERE**” *Boolean expression*) and the “**GROUP BY**” **expressions**.

### F.10.5 Can I see an example of a Transform operation?

A simple but quite complete example of a **Transform operation**<sup>61</sup> is:

```
TRANSFORM StDev(Temp_Max) AS GenVals
SELECT Capital AS Cap_City, Cal_Year AS C_Year
, StDev(Temp_Max) AS StDev_T_Max_Year
FROM T_Capital_temps
GROUP BY Capital, Cal_Year
PIVOT Quart
```

The Table “**T\_Capital\_Temps**” used in this example has the following structure and values:

T_Capital_Temps					
City	District	Cal_Year	Quart	Temp_max	Temp_min
Beijing	Dongcheng	2018	Q1	12.3	0
Beijing	Dongcheng	2018	Q2	4	1
Beijing	Dongcheng	2018	Q3	7.8	6.7
Beijing	Dongcheng	2018	Q4	17	15
Beijing	Xicheng	2018	Q1	2.26	-3.25
Beijing	Xicheng	2018	Q2	5.6	-4.5
Beijing	Xicheng	2018	Q3	30	25
Beijing	Xicheng	2018	Q4	18	13
Brasilia	Asa_Norte	2019	Q1	7.4	-0.96
Brasilia	Asa_Norte	2019	Q2	7.57	-1.05
Brasilia	Asa_Norte	2019	Q3	17.5	7.4
Brasilia	Asa_Norte	2019	Q4	10.04	2.35
Brasilia	Guara_I	2019	Q1	10.62	3.17
Brasilia	Guara_I	2019	Q2	11.43	4.21
Brasilia	Guara_I	2019	Q3	12.4	5.52
Brasilia	Guara_I	2019	Q4	11.12	3.81

<sup>61</sup> This is the Query “**F\_Transform\_examp**” from the “**Company\_Database.accdb**” file.

T_Capital_Temps					
City	District	Cal_Year	Quart	Temp_max	Temp_min
Washington	Anacostia	2018	Q1	6.67	-0.57
Washington	Anacostia	2018	Q2	7.55	-1.24
Washington	Anacostia	2018	Q3	7.65	-1.07
Washington	Anacostia	2018	Q4	7.68	-0.95
Washington	Downtown	2018	Q1	12.13	-9.55
Washington	Downtown	2018	Q2	5.67	-3.25
Washington	Downtown	2018	Q3	2.26	-4.3
Washington	Downtown	2018	Q4	17.57	-2.23
Washington	Bloomingdale	2018	Q1	3.15	2.13
Washington	Bloomingdale	2018	Q2	7.16	-1.92
Washington	Bloomingdale	2018	Q3	7.53	-1.58
Washington	Bloomingdale	2018	Q4	8.85	-0.9

Then, the **Transform** operation above would produce the following **output** record-list:

F_Transform_examp						
Cap_City	C_Year	StDev_T_Max_Year	Q1	Q2	Q3	Q4
Beijing	2018	9.28	7.10	1.13	15.70	0.71
Brasilia	2019	3.17	2.28	2.73	3.61	0.76
Washington	2018	3.98	4.52	0.99	3.08	5.40

Let me explain in more detail why are you getting this **output** record-list.

### What is the input record-list of this example?

The **input record-list** is the Table “**T\_Capital\_Temps**” in the “**FROM**” clause.

Since in this example there is no “**WHERE**” clause, the **retained input records** are **all** the records in the Table “**T\_Capital\_Temps**”.

### What is the output record-list of this example?

The **output** record-list of is determined by the “**GROUP BY**” clause. The “**GROUP BY**” **expressions** are “**Capital**” and “**Cal\_Year**”. Therefore, the **output** records correspond to the **distinct values** produced by the (retained) **input records** in the “**GROUP BY**” **expressions** “**Capital**” and “**Cal\_Year**”. If you check the Table “**T\_Capital\_Temps**” you will see that there are **three** distinct **arrays** of values for “**Capital**” and “**Cal\_Year**”:

Capital	Cal_Year
Beijing	2018
Brasilia	2019
Washington	2018

### What are the output field names of this example?

The **names** of the “**SELECT**” **fields** are “**Cap\_City**”, “**C\_Year**” and

“StDev\_T\_Max\_Year”, as indicated in the “SELECT” clause.

The **names** of the “PIVOT” **fields** are “Q1”, “Q2”, “Q3” and “Q4”, because these are the **distinct** results of the “PIVOT” **expression** “Quart”, when computed over **all** the (retained) **input records**.

### What is the output field order of this example?

The “SELECT” **fields** are produced in that order because this is how they appear in the “SELECT” clause.

The “PIVOT” **fields** are produced in that order because the data type of the “PIVOT” **expression** is *Strings*, and *Strings* are ordered in **alphabetical** order.

### What are the output field values of this example?

The **values** of the “SELECT” **fields** are the result of the “SELECT” **expressions** “Capital”, “Cal\_Year” and “StDev(Temp\_Max)” computed over the **groups** of (retained) **input records** produced by the “GROUP BY” **expressions**.

The (retained) **input records** in each **group** are the ones that produce the same values in the “GROUP BY” **expressions**. In this example, there are three **groups**, each characterized by the **results** “(Beijing, 2018)”, “(Brasilia, 2019)” and “(Washington, 2018)”. You may check that the **values** of the “SELECT” **fields** above correspond to the explanation I have just given.

The **values** of the “PIVOT” **fields** are the result of the “TRANSFORM” **expression** “StDev(Temp\_max)” computed over the **groups** of (retained) **input records** produced by the “GROUP BY” **expressions** jointly with the “PIVOT” **expression**.

In more detail, the **value** of field “j” from the **output record** “i” is produced by applying the “TRANSFORM” **expression** “Avg(Temp\_max)”, to **all** the records in the **group** “(i, j)” of (retained) **input records** corresponding to **record** “i” and **field** “j”. Each **group** “(i, j)” contains the (retained) **input records** that produce the **same results** in **all** the 1 to k “GROUP BY” **expressions** as the **values** in **row** “i” and the **same result** in the “PIVOT” **expression** as the **field name** of column “j”.

I want to point out **two relevant** aspects from this example, that apply to **all Transform** operations:

- Notice that the “SELECT” **expression** “StDev(Temp\_Max)” is **exactly the same** as the “TRANSFORM” **expression** “StDev(Temp\_Max)”. However, being the same expression it produces **different results** because it is computed over **different groups** of (retained) **input records**. The “SELECT” **expression** is computed over **groups** produced by the “GROUP BY” **expressions**, while the “TRANSFORM” **expression** is computed over **groups** produced the “GROUP BY” **expressions** jointly with the “PIVOT” **expression**.
- Notice that the field **values** of “StDev\_T\_Max\_Year” are **not** the result of SQL aggregate function “StDev( )” over the **values** of **fields** “Q1”, “Q2”, “Q3” and “Q4”, in **each row**. If they were computed like that, the result would be: “7.00”, “1.19” and “1.93, which is different from the actual **output** of the Query: “9.28”, “3.17” and “3.98”.

The reason is that the field **values** of “**StDev\_T\_Max\_Year**” are the result of the “**SELECT**” **expression** “**StDev(Temp\_Max)**” computed over the **groups** of (retained) **input records** produced by the “**GROUP BY**” **expressions**. In this example, this means computing the function “**StDev()**” over **all** the values of “**Temp\_Max**” for each “**Capital**” and each “**Cal\_Year**”.

### F.10.6 What is the “**TRANSFORM**” clause of a Transform?

In a **Transform** operation (click *F.10.1*), the mandatory “**TRANSFORM**” clause determines the “**PIVOT**” data/field **types** (click *F.10.3.4*) and the “**PIVOT**” **field values** (click *F.10.4.1*), as follows:

```
TRANSFORM PIVOT-field-values-expression()
```

If you want to write (syntax) a correct “**TRANSFORM**” clause, you may click *F.10.14*.

### F.10.7 What is the “**SELECT**” clause of a Transform?

In a **Transform** operation (click *F.10.1*), the mandatory “**SELECT**” clause determines the “**SELECT**” **fields** (click *F.10.3*), the “**SELECT**” **field names** (click *F.10.3.2*), the “**SELECT**” **field order** (click *F.10.3.3*), the “**SELECT**” data/field **types** (click *F.10.3.4*) and the “**SELECT**” **field values** (click *F.10.4.1*), as follows:

```
SELECT Output-expression_1() [ AS Output-field-name_1 ]
[ , ...
, Output-expression_n() [ AS Output-field-name_n ] ]
```

If you want to write (syntax) a correct “**SELECT**” clause, you may click *F.10.14*.

### F.10.8 What is the “**ORDER BY**” clause of a Transform?

In a **Transform** operation (click *F.10.1*), the optional “**ORDER BY**” clause determines the **order** of the **output** records, as follows:

```
ORDER BY [ Group_by-exp_x(Input-field-names) [DESC]
, ...
, Group_by-exp_y(Input-field-names) [DESC] ]
[[,] PIVOT-field-names-exp(Input-field-names) [DESC]]
```

Each “**ORDER BY**” **expression** must be **exactly the same** as one of the “**GROUP BY**” **expressions** or as the “**PIVOT**” **expression**, which is a very strong restriction.

Aside from this restriction on the **Transform** “**ORDER BY**” **expressions**, the “**ORDER BY**” of a **Transform** works exactly the same as the “**ORDER BY**” of a **Select**: you may click “*F.7.12 How do I use “ORDER BY” to order the output records of a Select?*”.

If you want to write (syntax) a correct “**ORDER BY**” clause, you may click *F.10.14*.

### F.10.9 What is the “**WHERE**” clause of a Transform?

In a **Transform** operation (click *F.10.1*), the optional “**WHERE**” clause indicates what are the **retained input records**, as follows:

```
WHERE Where-Boolean-expression(Input-field-names)
```

The “**WHERE**” *Boolean* expression is built using the “**Input-field-names**”



combining them with functions (excluding SQL aggregate), value operators and constants.

The “**WHERE**” clause of a **Transform** works **exactly the same** as the “**WHERE**” clause of a **Select**. You may see “*F.7.7 What is the “WHERE” clause of a Select?*”.

If you want to write (syntax) a correct “**WHERE**” clause, you may click *F.10.14*.

### **F.10.10 What is the “GROUP BY” clause of a Transform?**

In a **Transform** operation (click *F.10.1*), the mandatory “**GROUP BY**” clause determines the **output records** (click *F.10.4.2*), as follows:

```
GROUP BY      Group_by-expression_1 (Input-field-names)
             [ , ...
             , Group_by-expression_k (Input-field-names) ]
```

Each “**GROUP BY**” **expression** is built using the “**Input-field-names**” combining them with functions (excluding SQL aggregate), value operators and constants.

The “**GROUP BY**” of a **Transform** works **exactly the same** as the “**GROUP BY**” of a **Select**. You may click “*F.7.9 What is the “GROUP BY” clause of a Select-group by aggreg?*”.

There is also a very subtle, but very interesting, difference: the “**GROUP BY**” clause is optional in a **Select**, but it is mandatory in a **Transform**. This is because you can only do a cross table if there is **only one value** for each **row** and **column**. If you had **several values** for **each row** and **column**, there would be **no criterion** to choose one of them to be displayed. Therefore, the **Transform** operation **guarantees** that there is **only one value** for each **row** and **column**. The way to **guarantee** this is producing the cross table **values** from record **aggregation** of the **rows** and **columns**. Remind that record aggregation produces **only one value** from **each group** of records.

If you want to write (syntax) a correct “**GROUP BY**” clause, you may click *F.10.14*.

### **F.10.11 What is the “PIVOT” clause of a Transform?**

In a **Transform** operation (click *F.10.1*), the mandatory “**PIVOT**” clause determines the “**PIVOT**” **fields** (click *F.10.3*), the “**PIVOT**” **field names** (click *F.10.3.2*) and the “**PIVOT**” **field order** (click *F.10.3.3*), as follows:

```
PIVOT PIVOT-field-names-expression (Input-field-names)
```

The “**PIVOT**” clause contains the **expression** that produces the “**PIVOT**” **field names** (click *F.10.3.2*), **unless** the optional “**IN**” clause (click *F.10.12*) is used. If the optional “**IN**” clause is used, then the “**PIVOT**” **field names** are produced by the **list** of **values** in the “**IN**” clause.

The “**PIVOT**” **expression** is built using the “**Input-field-names**” combining them with functions (excluding SQL aggregate), value operators and constants.

If you want to write (syntax) a correct “**PIVOT**” clause, you may click *F.10.14*.



### F.10.12 What is the “IN” clause of a Transform?

In a **Transform** operation (click *F.10.1*), the optional “**IN**” clause determines the “**PIVOT**” **fields**, the “**PIVOT**” **field names** (click *F.10.3.2*) and the “**PIVOT**” **field order** (click *F.10.3.3*), as follows:

```
IN ( List-of-PIVOT-values )
```

The “**IN**” keyword is followed by a **list** of **constants**. I will call this **list** the “**IN**” **list**. The “**IN**” **list** is enclosed between parentheses and the **constants** are separated with commas.

The “**IN**” clause is most frequently used to specify the **order** of the “**PIVOT**” **fields** and/or to **discard** some of them. The way to do this is just by writing in the “**IN**” **list** the “**PIVOT**” **field names** that you want (i.e., excluding some if you do not want them), and writing them in the **specific order** that you want them.

In the general case, when the “**IN**” clause is used, the “**PIVOT**” **fields**, their **field names** and their **field order** will be **exactly the ones** and with the **same order** as they appear in the “**IN**” clause (except if they are the same as a “**SELECT**” **field name**). I now explain the possible cases in more detail:

- If the “**IN**” **list** **does not** include a **value** that is produced by the “**PIVOT**” **expression**, then the “**PIVOT**” **field** corresponding to that **value** **does not** appear in the **output** of the **Transform**.
- If the “**IN**” **list** **includes** a **value** that is produced by the “**PIVOT**” **expression**, and that **value** (converted to *String*) is **not** one of the “**SELECT**” **field names**, then the corresponding “**PIVOT**” **field** will be in the **output** of the **Transform**, in the order indicated in the “**IN**” **list**, with the corresponding **field values** produced by the “**TRANSFORM**” **expression**.
- If the “**IN**” **list** **includes** a **value** that is **not** produced by the “**PIVOT**” **expression**, and that **value** (converted to *String*) is **not** one of the “**SELECT**” **field names**, then that **value** (converted to *String*) will be a “**PIVOT**” **field**, in the order indicated in the “**PIVOT**” **list**, with all its values being **Null**.
- If the “**IN**” **list** **includes** a **value** that (converted to *String*) is **one of** the “**SELECT**” **field names**, then a “**PIVOT**” **field** will be produced with the name “**FieldN**”, where “**N**” is an integer value assigned by MS-Access. The order of that field is the one of the corresponding **value** in the “**IN**” **list**. The **values** of this field will be exactly the same ones as the ones of “**SELECT**” **field** whose **name** was the same.

The “**PIVOT**” **field name** corresponding to a **Null** value (i.e., “**PIVOT**” **field name** “<>”) can also be reordered by including it in the “**IN**” clause. I strongly recommend that you **do not use** a “**PIVOT**” **expression** that produces the string “<>”, to avoid confusion with a “**PIVOT**” **field name** arising from **Null** (which most likely will be unintentional).

The “**IN**” **list** **cannot** contain **duplicated** values: if it does, the Query will **crash** with a syntax error message.

As I indicated at the beginning of this section, the “**IN**” clause is most frequently used

to specify the **order** of the “**PIVOT**” **fields** and/or to **discard** some of them. As I have explained just above, the “**IN**” clause can **also** be used to add **copies** of “**SELECT**” **fields** and/or to **create new** “**PIVOT**” **fields**. However, the two features in the previous sentence do not seem very useful to me, because the added fields would have either replicated **values** or **Null** (respectively), in **all** the **output** records.

If you want to write (syntax) a correct “**IN**” clause, you may click *F.10.14*.

### **F.10.13 How do the clauses from Transform and Select compare?**

To better understand **Select** and **Transform**, I think it is useful to compare the characteristics of the clauses of a **Select** operation and of a **Transform** operation:

- In the **Transform** (mandatory) “**SELECT**” clause, each “**SELECT**” **expression** may include as its **elements** the “**PIVOT**” **expression**, in addition to the **elements** in the (mandatory) “**SELECT**” clause of a **Select-group\_by\_aggreg**. This allows to use the “**PIVOT**” **field names** in the “**SELECT**” **expressions**.  
Remind that in a **Select-group\_by\_aggreg**, the “**SELECT**” **expressions** can contain **other** “**Output-field-names**” (as long as you **do not** create a **circular** reference, click *F.7.14*), any number of “**GROUP BY**” **expressions**, and any number of **SQL aggregate functions** each having as argument its specific **expression** over the “**Input-field-names**” and **other** “**Output-field-names**” (as long as you **do not** create a **circular** reference, **nor** a **nested** SQL aggregate function, click *F.7.14*).
- The **Transform** (optional) “**DISTINCT**” clause does not produce any effect in the results of the **Transform** operation, while in the **Select** operation the (optional) “**DISTINCT**” clause removes **all** the redundant **output** duplicate records.
- The **Transform** (optional) “**ORDER BY**” clause can only use as its **expressions** the “**GROUP BY**” **expressions** (over the **Input-field-names**) or the “**PIVOT**” **expression** (over the **Input-field-names**). Aside from this, its functionality is the same as the (optional) “**GROUP BY**” clause from a **Select**.
- **Transform** does **not** have the (optional) “**HAVING**” and “**TOP**” clauses, that the **Select** operator has.
- **Select** does **not** have the (mandatory) “**TRANSFORM**” and “**PIVOT**” clauses nor the (optional) “**IN**” clause, that the **Transform** operator has.
- Both **Transform** and **Select** have the (**not advisable**) (optional) “**DISTINCTROW**” clause.

### **F.10.14 How do I write a correct (syntax) Transform?**

You may click:

- “*F.10.14.1 What is a syntax-example of a Transform?*”
- “*F.10.14.2 What are the formal rules (syntax) to write a Transform?*”
- “*F.10.14.3 Can I nest Transform operations?*”

### F.10.14.1 What is a syntax-example of a Transform?

An illustrative example of a fairly complete **Transform operation**<sup>62</sup> is:

```

TRANSFORM 5*Out_fld
          + Len("Q" & Quart)
          + Exp(Len(Capital))
          + Sum(1+Cal_Year) AS GenVals
SELECT    Log(2*Out_fld)           AS Nonsense_1
          , Len("T" & ("Q" & Quart)) AS Nonsense_2
          , 5*(2*Cal_Year)         AS Nonsense_3
          , 5*Max(Len(Quart & Capital)) AS Out_fld
FROM T_Capital_temps
GROUP BY Len(Capital), 2*Cal_Year
ORDER BY 2*Cal_Year DESC, Len(Capital)
PIVOT "Q" & Quart ;

```

The elements of the “**TRANSFORM**” **expression** are:

- The “**SELECT**” **field name** “**Out\_fld**”.
- The “**PIVOT**” **expression** “**"Q" & Quart**”.
- The first “**GROUP BY**” **expression** “**Len(Capital)**”.
- The SQL aggregate function “**Sum()**” computed over an expression that includes the **input field name** “**Cal\_Year**”.

There are **four** “**SELECT**” **expressions**:

- Each of the **four** “**SELECT**” **expressions** is assigned a “**SELECT**” **field name** using the optional “**AS**” clause. The four “**SELECT**” **field names** are “**Nonsense\_1**”, “**Nonsense\_2**”, “**Nonsense\_3**” and “**Out\_fld**”.
- The “**SELECT**” **expression** of field “**Nonsense\_1**” uses as an **element** the “**SELECT**” **field name** “**Out\_fld**”.
- The “**SELECT**” **expression** of field “**Nonsense\_2**” uses as an **element** the “**PIVOT**” **expression** “**"Q" & Quart**”.
- The “**SELECT**” **expression** of field “**Nonsense\_3**” uses as an **element** the second “**GROUP BY**” **expression**: “**2\*Cal\_Year**”.
- The “**SELECT**” **expression** of field “**Out\_fld**” uses as an **element** the SQL aggregate function “**Max()**” computed over an expression that includes the **input field names** “**Quart**” and “**Capital**”.

The **input record-list** in the “**FROM**” clause is:

- The Table name “**T\_Capital\_temps**” (click *F.10.5*).

The **two** “**GROUP BY**” **expressions** are:

- “**Len(Capital)**”, that uses as an **element** the **input field name** “**Capital**”.
- “**2\*(Cal\_Year)**”, that uses as an **element** the **input field name** “**Cal\_Year**”.

---

<sup>62</sup> This is the Query “**F\_Transform\_Syntax**” from the “**Company\_Database.accdb**” file.

The **two** “ORDER BY” expressions are (left to right):

- “**2\*(Cal\_Year)**” using the keyword “DESC” to indicate descending order. This **expression** is **exactly the same** as the second “GROUP BY” expression.
- “**Len(Capital)**” in ascending (default) order. This **expression** is **exactly the same** as the first “GROUP BY” expression.

The **elements** of the “PIVOT” expression are:

- The **input field name** “**Quart**”.

Let me point out a few relevant issues:

- If you remove the parentheses enclosing the “PIVOT” expression in “**"Q" & Quart**” from the second “SELECT” expression, the Query will produce a **syntax error**. This is because without the parentheses, the evaluation order of the expression evaluates first “**"T" & "Q"**”, and this causes that the “PIVOT” expression is not recognized.
- If you modify in any way the “ORDER BY” expressions (e.g., by adding “+1”), the Query will produce a **syntax error**. This is because each “ORDER BY” expression **must be exactly the same** as one of the “GROUP BY” expressions.
- If you add the “SELECT” expression “**Count(Out\_fld)**”, the Query will produce a **syntax error**. This is because you are using the “**Max()**” SQL aggregate function as an argument of the “**Count()**” SQL aggregate function, through the “SELECT” **field name** “**Out\_fld**”. You are therefore **nesting** SQL aggregate functions, which is not allowed.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

### F.10.14.2 What are the formal rules (syntax) to write a Transform?

A correct **Transform operation** has to be written in the following way:

```

TRANSFORM
  PIVOT-field-values-exp    (  Output-field-names
                               ,  PIVOT-field-names-exp(Input-field-names)
                               ,  Group_by-exp_1 (Input-field-names)
                               /  ...
                               ,  Group_by-exp_k (Input-field-names)
                               ,  SQL_agg_func(exp_t11(INOUT-field-names))
                               /  ...
                               ,  SQL_agg_func(exp_t1d(INOUT-field-names))
                               ) [ AS Output-values-Identifier ]

SELECT [ DISTINCT ] [ DISTINCTROW or ALL ]
        Output-exp_1(  Output-field-names
                      ,  PIVOT-field-names-exp(Input-field-names)
                      ,  Group_by-exp_1 (Input-field-names)
                      /  ...
                      ,  Group_by-exp_k (Input-field-names)
                      ,  SQL_agg_func(exp_o11(INOUT-field-names))
                      /  ...
                      ,  SQL_agg_func(exp_o1y(INOUT-field-names))
                      ) [ AS Output-field-name_1 ]
    [ , ...
      , Output-exp_n(  Output-field-names
                    ,  PIVOT-field-names-exp(Input-field-names)
                    ,  Group_by-exp_1 (Input-field-names)
                    /  ...
                    ,  Group_by-exp_k (Input-field-names)
                    ,  SQL_agg_func(exp_on1 (INOUT-field-names))
                    /  ...
                    ,  SQL_agg_func(exp_onz (INOUT-field-names))
                    ) [ AS Output-field-name_n ] ]

FROM {  [ ( { Table-name or Query-name } ) ] ]
        or [  { Table-name or Query-name }  [ AS Input-record-list-name ] ]
        or [ ( { Select-opr or Union-opr } ) [ AS Input-record-list-name ] ]
        or [ ( { Inner-or-Outer-Join-opr } ) or Cross-Join-opr ] }

[ WHERE Where-Boolean-exp (Input-field-names) ]

GROUP BY    Group_by-exp_1 (Input-field-names)
            [ , ...
              , Group_by-exp_k (Input-field-names) ]

[ ORDER BY [  Group_by-exp_x (Input-field-names) [ DESC ]
              /  ...
              , Group_by-exp_y (Input-field-names) [ DESC ] ]
            [ [ , ] PIVOT-field-names-exp (Input-field-names) [ DESC ] ] ]

PIVOT PIVOT-field-names-exp (Input-field-names)

[ IN ( List-of-PIVOT-values ) ]

```

The words in **bold** font are SQL **keywords**. The elements enclosed in square brackets “[ ]” are optional, and you may use each of these elements or not, depending on your desired result from the **Transform** operation. The elements separated with “or” are alternative options. Each list of alternative options for an element is enclosed between curly braces “{ }” when the element is **not optional** and between square brackets “[ ]” when the element is **optional**. Some curly braces “{ }” and square brackets “[ ]” are colored just to make it easier to see which ones are paired.

The elements in gray text are the ones I advise you do not use:

- I advise you **do not use** the optional “**AS**” clause for the “**TRANSFORM**” **expression** because it does not have any effect.
- I advise you **do not use** the optional “**DISTINCT**” clause because it does not have any effect.
- I advise you **do not use** the optional “**DISTINCTROW**” clause because it is not compatible with SQL server, and because you can get its same functionality in an easy way. If you want to know more, you may click “[F.7.8 What is the “DISTINCTROW” clause of a Select?](#)”.
- I advise you **do not use** the optional “**ALL**” clause because it does not have any effect.

All the terms above with a trailing “-exp” are **expressions**. The **terms** enclosed between the **parentheses** of each expression are the **elements** that can be used to build that specific expression, by **combining** these **elements** with functions (excluding SQL aggregate), value operators and constants.

All the terms above with a trailing “-opr” are SQL operations.

The “**Input-field-names**” are **all** the **input field names** from the **input record-list**.

The “**INOUT-field-names**” are the **1** to **v** “**Input-field-names**” and/or the **1** to **n** “**Output-field-names**”.

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click “[F.11.2 What are the SQL color codes used in this Guide?](#)”.

The **Transform** operation must be **the first** in a Query code, and a Query **can only contain** this first **Transform** operation. A **Transform** operation cannot be used as an **input record-list** in any other Query or expression. A Query that has a **Transform** operation **cannot** be used in any other Query. A **Transform** operation is therefore the **last** operation that can be performed over a record-list.

I will now explain how to write the different clauses of the **Transform** operation in the order they appear in the SQL code, because I think this is more convenient for syntax purposes.

### How do I write the “**TRANSFORM**” clause?

**TRANSFORM**

```

PIVOT-field-values-exp ( Output-field-names
    , PIVOT-field-names-exp (Input-field-names)
    , Group_by-exp_1 (Input-field-names)
    / ...
    , Group_by-exp_k (Input-field-names)
    , SQL_agg_func (exp_t11 (INOUT-field-names))
    / ...
    , SQL_agg_func (exp_t1d (INOUT-field-names))
) [AS Output-values-Identifier]

```

The “**PIVOT-field-values-exp()**” **expression** that you write after the “**TRANSFORM**” keyword is built over the **elements**:

- “**Output-field-names**” **1** to **n**



- “**PIVOT-field-names-exp (Input-field-names)**”  
This is, **exactly the same “PIVOT” expression.**
- “**Group\_by-exp (Input-field-names) 1 to k**”  
This is, any number of **exactly the same “GROUP BY” expressions.**
- “**SQL\_agg\_func (exp (INOUT-field-names))**”  
This is, any number of SQL aggregate functions (click *F.7.18*) each having as argument its specific **expression** over “**Input-field-names**” and/or “**Output-field-names**” (as long as you **do not** create a **nested SQL** aggregate function, click *F.7.14*).

You **cannot** assign an identifier (with an “**AS**” clause) neither to individual “**GROUP BY**” **expressions**, nor to the “**PIVOT**” **expression**. Therefore, you have to write them **as such** when you use them as **elements** to build the “**TRANSFORM**” **expression**. This may be cumbersome when a “**GROUP BY**” **expression** or the “**PIVOT**” **expression** is large, but there is no way to avoid this.

When you use the “**GROUP BY**” **expressions** and/or the “**PIVOT**” **expression** as **part** of a larger “**TRANSFORM**” **expression**, I strongly advise you to write **each** of the “**GROUP BY**” **expressions** and/or the “**PIVOT**” **expression** between parentheses to prevent that MS-Access rejects the larger “**TRANSFORM**” **expression** due to a different **evaluation order** than the one you expected (see the example in *F.7.16.1*).

Although you can use the “**Output-field-names**” as elements for the “**TRANSFORM**” **expression** and for the “**SELECT**” **expressions**, remind that you **cannot** use them for the “**ORDER BY**” **expressions**.

### How do I write the “**SELECT**” clause?

```
SELECT [DISTINCT] [DISTINCTROW or ALL]
      Output-exp_1( Output-field-names
                  , PIVOT-field-names-exp (Input-field-names)
                  , Group_by-exp_1 (Input-field-names)
                  , ...
                  , Group_by-exp_k (Input-field-names)
                  , SQL_agg_func (exp_o1l (INOUT-field-names))
                  , ...
                  , SQL_agg_func (exp_o1y (INOUT-field-names))
                  ) [ AS Output-field-name_1 ]
[ , ...
  , Output-exp_n( Output-field-names
                  , PIVOT-field-names-exp (Input-field-names)
                  , Group_by-exp_1 (Input-field-names)
                  , ...
                  , Group_by-exp_k (Input-field-names)
                  , SQL_agg_func (exp_onl (INOUT-field-names))
                  , ...
                  , SQL_agg_func (exp_onz (INOUT-field-names))
                  ) [ AS Output-field-name_n ] ]
```

The “**SELECT**” clause is written like in a **Select-group\_by\_aggreg**, but, using also the “**PIVOT**” **expression** as an element for the “**Output-exp\_i ()**” **expressions**.

After the “**SELECT**” keyword you write the “**SELECT**” **expressions** “**Output-exp\_i ()**” separated with commas “**,**”. **Each “SELECT” expression** that you write produces **one “SELECT” field**.



Each “**Output-exp<sub>i</sub>()**” is built over the **same elements** as the ones in the “**TRANSFORM**” **expression** (see the “**TRANSFORM**” clause slightly above for an explanation), with the **additional** restriction in this case that if using “**Output-field-names**” you **do not** create a **circular** reference (click *F.7.14*). Remind also the advice about enclosing **each** of the “**GROUP BY**” **expressions** and/or the “**PIVOT**” **expression** between parentheses, to avoid problems with expression evaluation order (see the example in *F.7.16.1*).

If you add after a “**SELECT**” **expressions** the keyword “**AS**” followed by a **name**, this will be the **name** of this “**SELECT**” **field**. If you do **not add** this “**AS**” clause, then you have two cases:

- If the “**SELECT**” **expression** consists **exactly** of a single **input field name**, then the “**SELECT**” **field name** will be that same **input field name**.
- Otherwise, MS-Access will assign to that “**SELECT**” **expression** the **field name** “**ExprXXXX**”, where “**XXXX**” is a four-digit integer number.

Be aware that an expression “**exp<sub>o</sub>()**” that is the argument of a given SQL aggregate function **cannot** contain any SQL aggregate function. This **cannot** happen in a direct manner, **nor through a reference** to another “**Output-field-name**”.

#### How do I write the “**FROM**” clause?

```
FROM {      [ ( {Table-name or Query-name} ) ]
or [ {Table-name or Query-name} ] [AS Input-record-list-name] ]
or [ ( {Select-opr or Union-opr} ) ] [AS Input-record-list-name] ]
or [ ( {Inner-or-Outer-Join-opr} ) ] or Cross-Join-opr ] }
```

The writing rules are **exactly the same** as in the **Select** operation.

Right after the “**FROM**” keyword you write the **input record-list**. The **input record-list** can be a Table name or a Query name or a **Join** operation or a **Select** operation or a **Union** operation. Notice that this is **different** from the writing rules (syntax) of the **Join** operation.

Let me clarify the rules on **parentheses** and “**AS**” clause, depending on what is the **input record-list** after the “**FROM**” keyword:

- **A Table name or Query name**  
It **may** be enclosed between parentheses, or, it **may** have an “**AS**” clause to assign to it a new name, but it **cannot have both**: you **cannot** enclose it in parentheses, **and also** have an “**AS**” clause. This writing rule is the **same** as the one in the **Join** operation.
- **A Select operation or Union operation**  
It **must** be enclosed between parentheses. It **may also** have an “**AS**” clause to assign to it a name. This writing rule is **different** from the one of the **Join** and **Union** operations.
- **A Join operation other than a Cross-Join**  
It **may** be enclosed between parentheses, and it **must not** have an “**AS**” clause. This writing rule is the same as the one in the **Join** operation.
- **A Cross-Join operation**  
It **must not** be enclosed between parentheses, and it **must not** have an “**AS**”

clause. This writing rule is the same as the one in the **Join** operation.

### How do I write the “WHERE” clause?

```
WHERE Where-Boolean-exp (Input-field-names)
```

The writing rules of the “**WHERE**” clause are **exactly the same** as in the **Select** operation.

The “**Where-Boolean-exp ()**” expression that you write after the “**WHERE**” keyword is built by combining the **input field names** with functions (**excluding** SQL aggregate), value operators and constants.

### How do I write the “GROUP BY” clause?

```
GROUP BY      Group_by-exp_1 (Input-field-names)
             [ , ...
             , Group_by-exp_k (Input-field-names) ]
```

The writing rules of the “**GROUP BY**” clause are **exactly the same** as in the **Select** operation.

Each of the **expressions** “**Group\_by-exp-i (Input-field-names)**” that you write after the “**GROUP BY**” keyword is built by combining the **input field names** with functions (**excluding** SQL aggregate), value operators and constants.

### How do I write the “ORDER BY” clause?

```
ORDER BY [   Group_by-exp_x (Input-field-names) [DESC]
           /   ...
           ,   Group_by-exp_y (Input-field-names) [DESC] ]
         [,] PIVOT-field-names-exp (Input-field-names) [DESC]
```

The “**ORDER BY**” keyword is followed by a list of **expressions** separated with commas. Each **expression** may be followed by the optional keyword “**DESC**”. Each of the **expressions** must be **exactly the same** as one of the “**GROUP BY**” **expressions** or as the “**PIVOT**” **expression**.

You **cannot** assign an identifier (with an “**AS**” clause) neither to individual “**GROUP BY**” **expressions**, nor to the “**PIVOT**” **expression**. Therefore, you have to write them **as such**. This may be cumbersome when a “**GROUP BY**” **expression** or the “**PIVOT**” **expression** is large, but there is no way to avoid this.

### How do I write the “PIVOT” clause?

```
PIVOT PIVOT-field-names-exp (Input-field-names)
```

The “**PIVOT-field-names-exp ()**” **expression** that you write after the “**PIVOT**” keyword is built by combining the **input field names** with functions (**excluding** SQL aggregate), value operators and constants.

### How do I write the “IN” clause?

```
IN ( List-of-PIVOT-values )
```

You write a “**List-of-PIVOT-values**” enclosed between parentheses after the “**IN**” keyword. This **list** is composed of one or more **constants** separated with commas. You can also include **Null** in the list. The data type of **all** the constants, either directly or through type conversion, **must** be same as the data type of the “**PIVOT**” **expression**.

**Final remark**

The **Transform** operation may be enclosed between parentheses.

**F.10.14.3 Can I nest Transform operations?**

**Transform operations cannot** be nested. The **Transform operation** must be **the first** operation in a Query, and a Query **cannot contain** any other **Transform operation**. Consequently, a **Transform operation cannot** be used as an **input record-list** in any other Query or expression. A Query that has a **Transform operation cannot** be used in any other Query. The **Transform** operator can therefore only be the **very last** operation over a record-list.

**F.11 What are the SQL clauses, their expression's elements and color codes?**

You may click:

- “F.11.1 Given an SQL clause, what are its expression's elements?”
- “F.11.2 What are the SQL color codes used in this Guide?”

**F.11.1 Given an SQL clause, what are its expression's elements?**

I list here **all** the SQL clauses, and for each of them what are the **elements** that can be combined with functions (**excluding** SQL aggregate), operators and constants to build the expression(s) associated to the clause. I list all the SQL clauses in alphabetical order:

“**ALL**” clause

It does not have an expression. My advice is you do not use this clause because it has no effect.

“**DISTINCT**” clause

It does not have an expression. You may click “F.7.11 What is the “DISTINCT” clause of a Select?”.

“**DISTINCTROW**”

It does not have an expression. My advice is you do not use this clause. You may click “F.7.8 What is the “DISTINCTROW” clause of a Select?”.

“**FROM**” clause

It does not have an expression (just the **input record-list**). You may click “F.7.4 What is the input record-list (“FROM” clause) of a Select?”.

“**GROUP BY**” expressions

Their elements are the “**Input-field-names**”. You may click “F.7.9 What is the “GROUP BY” clause of a Select-group by aggreg?”.

“**HAVING**” Boolean expression

This depends on the type of SQL operation where the “**HAVING**” clause is used, as

follows:

- **Select-total\_aggreg:**  
The “**HAVING**” *Boolean expression* elements can be any number of SQL aggregate functions each having as argument its specific **expression** over “**Input-field-names**”.
- **Select-group\_by\_aggreg:**  
The “**HAVING**” *Boolean expression* elements can be the “**GROUP BY**” **expressions** (over “**Input-field-names**”) plus any number of SQL aggregate functions each having as argument its specific **expression** over the “**Input-field-names**”.

You may click “[F.7.10 What is the “HAVING” clause of Select-group\\_by\\_aggreg or Select-total\\_aggreg?](#)”.

“**IN**” clause

It does not have an expression (just a **list** of **values**). You may click “[F.10.12 What is the “IN” clause of a Transform?](#)”.

“**ON**” *Boolean expression*

It is composed of one or more individual *Boolean* expressions composed with *Boolean* operators. **Each** individual *Boolean* expression must include **at least one qualified input field name** from **each** of the two **input** record-lists. You may click “[F.8.9 What is the “ON” clause of “INNER JOIN” and Outer-Join \(“LEFT JOIN” and “RIGHT JOIN”\)?](#)”.

“**ORDER BY**” expressions

This depends on the type of SQL operation where the “**ORDER BY**” clause is used and on the usage of the “**DISTINCT**” clause, as follows:

- **Any Select** with the “**DISTINCT**” clause:  
Each “**ORDER BY**” expression **must be exactly the same** as one of the “**SELECT**” **expressions** and **cannot** contain any “**Output-field-name**”.
- **Select-no\_aggreg** without the “**DISTINCT**” clause:  
Their elements can be the “**Input-field-names**”.
- **Select-group\_by\_aggreg** without the “**DISTINCT**” clause:  
Their elements can be any number of the “**GROUP BY**” **expressions** (over “**Input-field-names**”) and any number of SQL aggregate functions each having as argument its specific **expression** over the “**Input-field-names**”.

You may click “[F.7.12 How do I use “ORDER BY” to order the output records of a Select?](#)”.

- **Transform:**  
Each “**ORDER BY**” expression **must be exactly the same** as one of the “**GROUP BY**” **expressions** (over “**Input-field-names**”). You may click “[F.10.8 What is the “ORDER BY” clause of a Transform?](#)”.

**“PIVOT” expression**

Its elements are the **“Input-field-names”**. You may click [“F.10.11 What is the “PIVOT” clause of a Transform?”](#).

**“SELECT” expressions**

This depends on the type of SQL operation where the **“SELECT”** clause is used, as follows:

- **Select-no\_aggreg:**  
The **“SELECT” expression elements** can be the **“Output-field-names”** and the **“Input-field-names”**. You may click [“F.7.6.1 What are the output field values of a Select-no aggreg?”](#).
- **Select-total\_aggreg:**  
The **“SELECT” expression elements** can be the **“Output-field-names”** and any number of SQL aggregate functions each having as argument its specific **expression** over the **“INOUT-field-names”**. You may click [“F.7.6.3 What are the output field values of a Select-total aggreg?”](#).
- **Select-group\_by\_aggreg:**  
The **“SELECT” expression elements** can be the **“Output-field-names”**, the **“GROUP BY” expressions** (over **“Input-field-names”**) and any number of SQL aggregate functions each having as argument its specific **expression** over the **“INOUT-field-names”**. You may click [“F.7.6.2 What are the output field values of a Select-group\\_by aggreg?”](#).
- **Transform:**  
The **“SELECT” expression elements** can be the **“Output-field-names”**, the **“GROUP BY” expressions** (over **“Input-field-names”**), any number of SQL aggregate functions each having as argument its specific **expression** over the **“INOUT-field-names”** and the **“PIVOT” expression** (over **“Input-field-names”**). You may click [“F.10.3 What are the output fields of a Transform?”](#).

**“TOP”** or **“TOP PERCENT”** clause

It does not have an expression (just an integer value). You may click [“F.7.13 What is the “TOP” clause of a Select?”](#).

**“TRANSFORM” expression**

Its elements can be the **“Output-field-names”**, the **“GROUP BY” expressions** (over **“Input-field-names”**), any number of SQL aggregate functions each having as argument its specific **expression** over the **“INOUT-field-names”** and the **“PIVOT” expression** (over **“Input-field-names”**). You may click [“F.10.4 What is the output record-list of a Transform?”](#).

**“WHERE” Boolean** expression

Its elements are the **“Input-field-names”**. You may click [“F.7.7 What is the “WHERE” clause of a Select?”](#).

If you want to know the SQL **color codes** used in this **Lightning Guide**, you may click [“F.11.2 What are the SQL color codes used in this Guide?”](#).

### F.11.2 What are the SQL color codes used in this Guide?

Along *Part F* I use **color codes** to better identify the **elements** used to build the **expressions** in the different SQL **clauses**. These **same** color codes are also used in other Parts of this **Lightning Guide**. However, be aware that along **other Parts** I may also use the same colors for other purposes: I think that when this happens it is pretty clear from context, but just in case I want to say it explicitly.

Each **expression** belonging to each **specific** SQL clause is built combining its **corresponding elements** with functions (**excluding** SQL aggregate), operators and constants.

The following list indicates, for **each color**, what are the **elements** depicted with **that color**. Colors are listed in alphabetical order:

#### Blue (dark)

In **Transform** operations, the **functions** (**excluding** SQL aggregate), **operators** and **constants** of the “**TRANSFORM**” **expression**, the **name** of the “**TRANSFORM**” **expression**, and the **values** of the “**PIVOT**” **fields**. Notice that the **elements** of the “**TRANSFORM**” **expression** will usually **keep** their own colors.

#### Blue (light)

In **Join** operations, the **right input** field **names**, the **right input** “**Table-name**”, the **right input** “**Query-name**” and the **right input** record-list **name**.

In **Union** operations, the **right input records** and the **keywords** of the **right input** SQL operation.

In **Join** and **Union** operations, the generic (i.e., neither left, nor right) **input** fields and **input** records are colored partly in **maroon** and partly in **light blue**.

#### Brown

In all **Select**, **Join** and **Union** operations, the **output** fields, the “**Output-field-names**”, the **output** records, the **output** record-lists, and also the **functions** (**excluding** SQL aggregate), **operators** and **constants** of the “**SELECT**” **expressions**. Notice that the individual **elements** of the “**SELECT**” **expressions** will usually **keep** their own color codes.

In **Transform** operations, “**SELECT**” **field names**, and also the **functions** (**excluding** SQL aggregate), **operators** and **constants** of the “**SELECT**” **expressions**. Notice that the individual **elements** of the “**SELECT**” **expressions** will usually **keep** their own color codes.

#### Fuchsia

In **Select-group\_by-aggreg**, **Select-total\_aggreg** and **Transform** operations, the SQL aggregate functions.

#### Gold

The Query **parameters**, regardless of being declared or undeclared.

#### Green

In all **Select** and **Transform** operations, the **input** fields, the “**Input-field-names**”,



the **input** records, the **input** record-lists, the **input** “**Table-name**”, the **input** “**Query-name**” and the “**Input-record-list-name**” in the “**FROM**” clause.

In SQL code examples, the SQL “**-- comments**”.

### Maroon

In **Join** operations, the **left input** field **names**, the **left input** “**Table-name**”, the **left input** “**Query-name**” and the **left input** record-list **name**.

In **Union** operations, the **left input records** and the **keywords** of the **left input** SQL operation.

In **Join** and **Union** operations, the generic (i.e., neither left, nor right) **input** fields and the **input** records are colored partly in **maroon** and partly in **light blue**.

### Orange

In **Transform** operations, the “**PIVOT**” **expression**, the “**PIVOT**” **fields names** and the “**List-of-PIVOT-values**” in the “**IN**” clause. Notice that the color of the **elements** of the “**PIVOT**” **expression** (which are the “**Input-field-names**”) depends on context:

- In SQL code examples, the “**Input-field-names**” of the “**PIVOT**” **expression** are **also** colored in **orange**, to make it easier to visually identify this **expression** within **other** expressions of the **Transform** clauses.
- In “dummy” SQL code for syntax descriptions, the “**Input-field-names**” of the “**PIVOT**” **expression** keep their **green** color, to highlight what are the allowed elements for this **expression**.

### Purple

In **Select-group\_by-aggreg** and **Transform** operations, the “**GROUP BY**” **expressions**. Notice that the color of the **elements** of the “**GROUP BY**” **expressions** (which are the “**Input-field-names**”) depends on context:

- In SQL code examples, the “**Input-field-names**” of the “**GROUP BY**” **expressions** are **also** colored in **purple**, to make it easier to visually identify these **expressions** within **other** expressions of the **Select-group\_by-aggreg** and **Transform** clauses.
- In “dummy” SQL code for syntax descriptions, the “**Input-field-names**” of the “**GROUP BY**” **expressions** keep their **green** color, to highlight what are the allowed elements for these **expressions**.

### Turquoise

In all consulting operations, the “**INOUT-field-names**”, which represent the “**Input-field-names**” and/or the “**Output-field-names**”.

## F.12 How do I add parameters (type-in variables) to my Queries?

Just by using one or more **variable name(s)** as **parameter(s)**. However, it is **much better** practice to **declare them** explicitly using the “**PARAMETERS**” declaration.



This is an example of a Query with the “**PARAMETERS**” declaration:

```
PARAMETERS [Desired_City] Text (255), [Desired_District] Text (255) ;

SELECT City, District, Cal_Year, Quart, Temp_min, Temp_max
FROM T_Capital_Temps
WHERE (City = Desired_City) AND (District = Desired_District)
```

In this Query, the values of the parameters “**Desired\_City**” and “**Desired\_District**” are typed-in by the user each time that the Query is run, and each appearance of each parameter in the Query code is replaced by the value that the user typed-in. This Query would also work without the “**PARAMETERS**” declaration, but it is much better SQL programming practice to **always** declare **all** the parameters explicitly.

A **parameter** is a **variable** within the Query code whose value is **typed-in** by the user **each time** that the Query is **run**. The Query can then behave in some different way every time you run it. **Parameters** are to Queries like arguments to VBA functions, and they are extremely useful. Imagine you have the Table of temperatures by capital city and district (click *F.10.5*), but larger, with 150 cities, having 15 districts each, and over 12 years. This implies  $150 \times 15 \times 12 \times 4$  records, which is 108,000 records. Going manually through such a long Table may be tedious, so you want a Query to instantly provide you the records from the **city**, **district** and **calendar year** that you want. This is trivial to do with a parametrized Query.

The “**PARAMETERS**” declaration is the first statement of your Query code, and it is followed by the **list of variable names** that you want to use as **parameters**, each variable name followed by its **data type**, and ending the list with a semicolon “;”. Each variable name **must** be enclosed in **square brackets** and followed by the **data type** of the variable.

The correct way to write the “**PARAMETERS**” declaration is:

```
PARAMETERS [Name_1] Data_Type_1, ..., [Name_n] Data_Type_n ;
{Select-operation or Transform-operation or Union-operation} ;
```

All the **parameter names** “**Name\_1**”, ..., “**Name\_n**” **must be different**, but the **data types may repeat** as needed. Notice that in this particular case, the **square brackets** are **not** used to denote optional or alternative elements, and the **square brackets** are actually **part of the syntax** and **are mandatory**. For this reason, in this case I am using **curly braces** to enclose optional or alternative elements. Text shaded in gray denotes options that are valid but not recommended.

The **data types** you can assign to each variable of the “**PARAMETERS**” declaration are **neither** the same as the **Table field types**, **nor** the same as the **VBA data types**, although they are very similar. The following list presents the **specific** data types (written in **courier** font) that you can assign to each **parameter** in the “**PARAMETERS**” declaration, indicating for each of them its equivalent VBA data type (written in **bold italics**, as usual).

- **Text { (n) }**: **String** VBA data type.  
The part enclosed in curly braces is **optional**. The **integer** number “**n**” **between parentheses** indicates the maximum number of characters that the text string may have. Usual practice is to have **n=255**.

- **Bit:** *Boolean* VBA data type
- **Byte:** *Byte* VBA data type
- **Short** or **Int** or **Smallint:** *Integer* VBA data type
- **Integer** or **Long:** *Long* VBA data type
- **Real** or **Single:** *Single* VBA data type
- **Float** or **Double:** *Double* VBA data type
- **Date** or **Time** or **DateTime:** *Date* VBA data type (i.e., date **and** time value)
- **Binary:** Binary encoded number.

MS-Access will check that the value you enter **matches** the corresponding data type. If what you enter **does not match** the data type, MS-Access will show you an informative message and will allow you to enter the **parameter** again. Examples of data type **mismatch** are entering a string in a numeric **parameter** or entering a wrong date (e.g., 30<sup>th</sup> of February) in a date **parameter**.

Notice there is no data type equivalent to Table field type **Large Number**, nor to VBA **LongLong** data type.

In case you have **different parameter names** in two or more nested Queries (i.e., Queries that use one another), when running the outermost Query, the user will be prompted to type-in **all** the different **parameters** from **all** the nested Queries.

In case you have **the same parameter name, and** with a **compatible** data type, in several nested Queries (i.e., Queries that use one another) they will **all** be considered **the same parameter**, and the user will be prompted only once for it.

In case you have **the same parameter name, and** with **incompatible** data types, in two or more nested Queries (i.e., Queries that use one another) the Query will **crash** showing the error message:

“Wrong data type for parameter '[Param\_name]'.”

where “*Param\_name*” is the name of the **parameter** causing the error.

In this context, the *String* data type is **compatible** only with the *String* data type, all **numeric-like** data types are **compatible** among themselves, and the *String* data type is not compatible with any of the **numeric-like** data types.

When you run a Query with **parameters**, typing-in the **value** of a **parameter** is similar to typing-in the value of a Table field. If you want to know more about typing-in values, you may click “[E.2.2 How do I type-in a value in a field?](#)”.

## F.13 How do I write a Query that changes my Table data?

You may click:

- “[F.13.1 What is a Delete operation and how do I write it?](#)”
- “[F.13.2 What is an Insert operation and how do I write it?](#)”
- “[F.13.3 What is an Update operation and how do I write it?](#)”

- “F.13.4 Can I write a VBA function that deletes, inserts or updates Table records?”
- “F.13.5 When should I use SQL operations that change records from my Tables?”

### F.13.1 What is a Delete operation and how do I write it?

This section also answers the question:

- **How do I write a Query that deletes Table records?**

A **Delete** operation **deletes** all the records from the Table “Table\_name” that produce **True** in its “**WHERE**” **Boolean** expression. The **Delete** operation must be the only expression in the Query. The “**WHERE**” expression may make use of **all** the Table fields.

The way to write (syntax) a **Delete** operation is:

```
DELETE [T_name.*]
FROM Table_name
WHERE Boolean-expression(Table_field_names) ;
```

The elements between square brackets “[ ]” are optional.

Let me show an example based on the Table “T\_Capital\_Rainfall\_Q”:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

Copy this Table (right-click on it and click on “**Copy**” from the pop-up menu). Then paste it (right-click anywhere on the “**Navigation Pane**” and click on “**Paste**” from the pop-up menu), typing-in “T\_Insert\_Delete” as the Table name. You can then write the following Query<sup>63</sup>:

```
DELETE
FROM T_Insert_Delete
WHERE Capital="Beijing" ;
```

If you run this Query, it will delete all records where the value of Capital is “Beijing” from the Table “T\_Insert\_Delete”.

### F.13.2 What is an Insert operation and how do I write it?

This section also answers the question:

- **How do I write a Query that inserts Table records?**

<sup>63</sup> This is the Query “F\_Delete” from file “Company\_Database.accdb”.

You write a Query that inserts records into a Table using the “**INSERT INTO**” SQL operator. There are **two types** of Query that can be written with the “**INSERT**” operator.

The **first** type allows to insert **many** Table **records**, taking their values from a **Table**, a **Query** or a **Join operation**. This type is called the **Insert-many-records** operation:

- “*F.13.2.1 How do I write a Query that inserts many Table records?*”

The **second** type allows to insert **only one record**, with an **explicit list of field values**. This type is called the **Insert-one-record** operation:

- “*F.13.2.2 How do I write a Query that inserts only one Table record?*”

In either case, the **Insert** operation **must** be the **only** SQL operation in the Query.

If you want additional information, you may click:

- “*F.13.2.3 What are the common characteristics to all Insert operations?*”
- “*F.13.2.4 What is the summary of Insert operations?*”

### **F.13.2.1 How do I write a Query that inserts many Table records?**

The way to write (syntax) the **Insert-many-records** operation is:

```
INSERT INTO target_T_name [ ( Targ_Table_field_name_1
                             [, ...
                             [, Targ_Table_field_name_n] ) ]
[ IN externaldatabase]
<Select-operation>
```

The elements between square brackets “[ ]” are optional. The elements separated with “or” are alternative options.

The term “<Select-operation>” represents a **Select** operation **embedded** in the **Insert-many-records** operation. If you want to know the writing rules (syntax) of a **Select** operation, you may click “*F.7.14 How do I write a correct (syntax) Select?*”.

The **Insert-many-records** operation **attempts** to insert into the **target** Table “**target\_T\_name**” **all** the records produced by its **embedded Select** operation.

The **Insert-many-records** operation behaves **differently** depending on the **inclusion**, or not, of the **optional list of field names** from the target Table. This optional list is the list of field names that is placed after “**target\_T\_name**”.

- If you **include** the **optional list of target Table field names**  
The **matching** of **Select output values** to the **list of target Table field names** is based on their **left to right ordering**. This is, the **leftmost Select value** is assigned to the **leftmost** field name in the **list of target Table field names**, and so on. The **number of Select values** **must** be the same as the **number** of fields in the **list of target Table field names**.  
Notice that the **order** of field names in the **list of target Table field names** may be **completely different** from the order of fields in the **target Table**.  
Notice that the **Select output field names** **have no effect** and can be whatever you want, or you can just not assign **output field names**.
- If you **omit** the **optional list of target Table field names**  
The matching of **Select output values** to target Table field names is done based on the **field name**. This is, each **Select output value** is assigned to the target Table field

name with its **same field name**. This implies that **each and every Select output field name must exist** in the target Table: otherwise, the Query will **crash** showing the error message:

*“The INSERT INTO statement contains the following unknown field name: 'Field\_name'. make sure you have typed the name correctly, and try the operation again.”*

where “*Field\_name*” is the field name that caused the error.

The **Select output fields must be a subset of, or equal to**, the field names of the target Table.

The **order of Select output fields has no effect** and can be whatever you want.

If the target Table has one, or more, fields that do not match the **Select values**, then all the records to be inserted will have the **default value** in each of the **non-matching** field(s). If one, or more, of the **non-matching** fields **do(es) not** have a default value, then all the records to be inserted will have **Null** in these field(s).

Let me show an example of an **Insert-many-records** operation based on the same Tables “T\_Insert\_Delete” and “T\_Capital\_Rainfall\_Q” from *F.13.1*. You write the following Query<sup>64</sup>:

```
INSERT INTO T_Insert_Delete
SELECT Capital, Cal_Year, Quart, Quart_Rainfall
FROM T_Capital_Rainfall_Q
WHERE Capital="Beijing" ;
```

You then manually delete all the records in “**T\_Insert\_Delete**”, and then you **run** the above Query (“**F\_Insert**”). You will see that the Query has inserted all the records from “**T\_Capital\_Rainfall\_Q**” where “**Capital**” is “**Beijing**”.

My advice is you **always omit** the **list of target Table field names** in the **Insert-many-records** operation. The reason is that it is less error-prone to do the association of values based on **field name** than doing it by their **order** in the **list of target Table field names**.

### **F.13.2.2 How do I write a Query that inserts only one Table record?**

The way to write (syntax) the **Insert-one-record** operation is:

```
INSERT INTO target_T_name [ ( Targ_Table_field_name_1
                             [, ...
                             [, Targ_Table_field_name_n ] ) ]
VALUES ( exp_1() [, exp_2() [, exp_n()] ] )
```

The elements between square brackets “[ ]” are optional. The parentheses are **not** optional.

The **Insert-one-record** operation attempts to insert into the **target** Table “**target\_T\_name**” **only one record** whose field values are the ones returned by the “**VALUES**” expressions (i.e., the expressions found after the “**VALUES**” keyword).

The **Insert-one-record** operation **always** performs the matching of the returned values from its “**VALUES**” expressions based on their **left to right position**, regardless of using or not the optional **list of target Table field names**. The effect of using or not the

<sup>64</sup> This is the Query “**F\_Insert**” from file “**Company\_Database.accdb**”.

**optional list of target Table field names** is the following:

- If you **include** the **optional list of target Table field names**  
The result from the **leftmost “VALUES” expression** is assigned to the **leftmost** field name from the **list of target Table field names**, and so on.

Target Table fields not included in the **list of target Table field names** do not match any of the **“VALUES” expressions**. Therefore, all the records to be inserted will have the **default value** in each of these target Table fields **not** included in the **list**. If one, or more, of the **non-included** fields **do(es) not** have a default value, then all the records to be inserted will have **Null** in these field(s).

- If you **omit** the **list optional of target Table field names**  
The result from the **leftmost “VALUES” expression** is assigned to the **leftmost** field name in the **target Table**, and so on.

The **number** of **“VALUES” expressions** **must** be the same as either the **number** of field names in the **optional list** (when it exists), or the **number** of field names in the target Table (when the **optional list** does not exist). If the **number** is **not** the same, MS-Access does not allow you to save the Query, showing the syntax **error** message:

*“Number of query values and destination fields are not the same.”*

My advice is you **always include** the **list of target Table field names** in the **Insert-one-record** operation. The reason is that it is less error-prone to do the association of values based on the field name order in the **list of target Table field names** than on the Table field order. Also, if you change the field order in the Table, the **Insert** operation will work fine, while it will fail if you omit the **optional list of target Table field names**.

### **F.13.2.3 What are the common characteristics to all Insert operations?**

All **Insert** operations share the following characteristics:

- An **Insert** operation **must** be the **only** SQL operation in the Query.
- If you use the **optional list of target Table field names**, then the **field names** included in that **list must** be a **subset** of, or **equal** to, the **target Table field names**. Otherwise, the Query will **crash** showing the syntax **error** message:

*“The INSERT-INTO statement contains the following unknown field name: 'Field\_name'. Make sure you have typed the name correctly, and try the operation again.”*

where **“Field\_name”** is the field name included in the optional list of target Table field names that is **not** one of the field names of the target Table.

- If you use the **optional list of target Table field names**, then the **number of field names** in that list **must** be the same as either the **number** of **Select values**, or the **number** of **“VALUES” expressions**. If the **number** is **not** the same, then MS-Access does not allow you to save the Query, showing the syntax error message:

*“Number of query values and destination fields are not the same.”*

- If the **data type** of one, or more, **field values to be inserted** does not match the **field type** of its corresponding target Table field, MS-Access will attempt to do a type conversion. If a given type conversion is feasible (e.g., the text string **“123”** into a

**Number** field), the target field will have the corresponding value. If a given type conversion is **not** feasible (e.g., the text string “**Chicago**” into a **Number** field), the target field will have a **Null** (i.e., it will **not** have the default field value).

- If the number of values (either the **number** of **Select values**, or the **number** of “**VALUES**” expressions) is less than the number of target Table fields, then each non-matching target Table field will be set to its default value as configured in the target Table. If any such field does not have a configured default value, it will be set to **Null**. Remind that the **Insert-one-record** operation without the optional list of target Table field names **must** always insert all the Table fields, while the other types of **Insert** operation may insert less fields than the number of fields in the target Table.
- If any expression returns **Null** or an **exception-value**, then **Null** is placed into the corresponding field of the record to be inserted.
- MS-Access will perform, on **each and every record to be inserted**, the usual checks (i.e., the field/record validation rules, indexing values, master field values and Required fields). If you want to know more about the checks that MS-Access will perform, you may click “*E.6.1 What checks are done when saving a field value?*”

When you attempt to **save** (click Part E) a field value, MS-Access will always do the following **checks**:

- Check that the value **complies** the **field type** and **size**  
Click “*L.4.2.1 How do I fix an invalid value?*”.
- If the field is configured as “**Required=Yes**” (click D.5.1.7), check that its value is **non-Null**. Remind that **all Key fields** are **always** configured as “**Required=Yes**”.  
Click “*L.4.2.2 How do I fix trying to save Null in a “Required” field?*”.
- If it is a **Short Text** field configured as “**Allow Zero Length=No**” (click D.5.2.1), check that its value is **not** a string of zero length.  
Click “*L.4.2.3 How do I fix violating “Allow Zero Length=No”?*”.
- If **pasting** a text string, check that the length of the string is lesser or equal than the “**Field Size**” property of the **Short Text** field.  
Click “*L.4.2.4 How do I fix that I cannot paste a text string in full?*”.
- If you chose the value using a **drop-down menu** configured to only accept values from the menu (i.e., “**Limit to List=Yes**”), check that the value exists in the drop-down menu. Notice that this check **will not** be done when **pasting** values.  
Click “*L.4.2.5 How do I fix a value rejected because it is not in the list?*”.
- If the field has a **validation rule** (click D.5.1.5), check that its **Boolean** expression does not return **False**. Recall from D.5.1.5 that if the **Boolean** expression returns **Null**, the field value is considered **valid**.  
Click “*L.4.2.6 How do I fix a value violating a field validation rule?*”.

If **one or more** of the above checks is **wrong**, MS-Access shows an error **message** and/or requests you to **fix** the field value. The way MS-Access does this depends on whether you are:

- **Editing the record under edition** or **pasting** over a rectangle of field values over **one** record (click E.1).



- **Pasting** several rows of values as new records or **pasting** over a rectangle of field values over **several** records (click L.4.4).
- **Searching** and **replacing** data in records (click E.7.1).
- **Cascade updating slave fields** (click L.4.3.4).
- **Inserting** or **updating** records with an SQL Query.
- What checks are done when entering or modifying a record?. Every record that does not comply with one or more checks, will be dropped and will not be inserted into the target Table. For example, if a Table field cannot take **Null** (e.g., because it is a field configured as “**Required=Yes**”), the Query will work, but the records with **Null** in that field **will not** be inserted.
- MS-Access will show a message indicating how many records have been inserted, and how many have been discarded. Discarded records **are not** inserted in an automatically created Table, as it happens with discarded records in a paste operation, that are inserted in the “**Paste Errors**” Table.

### F.13.2.4 What is the summary of Insert operations?

The following table summarizes the main characteristics of **Insert** operations:

Type of Insert operation	With list of target Table field names	Without list of target Table field names
Many-records	Matching of <b>Select values</b> to fields in the <b>list</b> done by <b>order</b>	Matching of <b>Select values</b> to fields in the <b>target Table</b> done by <b>field name</b>
	<b>Select output field names</b> are irrelevant	<b>Select output field names must be a subset of, or equal to,</b> the target Table field names
	<b>Order of Select values</b> matters	<b>Order of Select values</b> irrelevant
	<b>Same number of Select values</b> as fields in the <b>list</b>	<b>Not applicable</b> (No <b>list</b> of Table field names)
	<b>Less than or equal number of Select values</b> as target Table field names	
	<b>Non-matching target Table fields</b> get <b>default</b> value or <b>Null</b>	
One-record	<b>Matching of “VALUES” expressions</b> to fields done by <b>order</b>	
	<b>Order of “VALUES” expressions</b> matters	
	Matching of <b>“VALUES” expressions</b> to fields in the <b>list</b>	Matching of <b>“VALUES” expressions</b> to fields in the <b>target Table</b>
	<b>Cannot assign field names</b> to <b>“VALUES” expressions</b>	
	<b>Same number of “VALUES” expressions</b> as fields in either the <b>list</b> or the <b>target Table</b>	
	Target Table <b>fields not in list</b> get <b>default</b> value or <b>Null</b>	<b>All target Table fields must match</b> a <b>“VALUES” expression</b>
Both of them	<b>Must be the only SQL operation</b> in the Query	
	<b>Field names in list</b> must be a <b>subset of, or equal to,</b> the target Table field names	<b>Not applicable</b> (No <b>list</b> of Table field names)
	<b>Same number of either Select values, or “VALUES” expressions,</b> as fields in the <b>list</b>	No common behavior.
	<b>Data type mismatch</b> between the value to insert and the Table field type causes the insertion of <b>Null</b> in the field.	
	<b>Non-matching target Table fields</b> get <b>default</b> value or <b>Null</b>	
	A <b>“SELECT”</b> or <b>“VALUES” expression</b> returning <b>Null</b> or exception value results in <b>Null</b> in the corresponding target Table field	
	Records that <b>do not comply</b> with the Table properties are <b>not inserted</b> .	

### F.13.3 What is an Update operation and how do I write it?

This section also answers the question:

- **How do I write a Query that modifies Table records?**

The **Update** operation is like a **batch** “search-and-replace” operation performed over Table records.

The way to write (syntax) an **Update** operation is:

```
UPDATE Table_name
SET   Field_name_1 = exp_1(Table_field_names)
    [ ...
      , Field_name_n = exp_n(Table_field_names) ]
WHERE Boolean-exp(Table_field_names) ;
```

The **Update** operation **modifies** the values of the fields “**Field\_name\_1**” to “**Field\_name\_n**” from those records of the Table “**Table\_name**” that produce *True* in the “**WHERE**” *Boolean* expression. For every **matching** Table record, the current value of **each** of its fields “**Field\_name\_1**” to “**Field\_name\_n**” is **replaced** by the **result** of the **corresponding** expression “**exp\_i()**” computed over **all** the field values of the **matching** record.

### F.13.4 Can I write a VBA function that deletes, inserts or updates Table records?

You can invoke SQL code from your VBA functions and therefore, you can write VBA functions that invoke **Delete** operations, **Insert** operations and/or **Update** operations.

Every time you invoke your user-defined VBA function, the corresponding operations (**Delete**, **Insert** and/or **Update**) will be performed over your database Tables.

### F.13.5 When should I use SQL operations that change records from my Tables?

I advise you **do not use** SQL operations that change records over your “**normal**” database Tables, because it is **extremely risky**. This applies both to using **data-changing** SQL operations in a Query or in VBA function.

I advise you **only use data-changing** SQL operations over Tables whose content is **automatically** generated by other **data-changing** SQL operations. These are typically auxiliary Tables and/or temporal Tables used in your Queries.

## F.14 How do I write and debug my SQL Queries?

If you want to know how to **edit** your SQL code, you may click “*F.4.4 What SQL Query editor should I use?*”.

If you want to know how to **correctly** write your SQL Query code, you may click “*F.6 What are the SQL operators I use to write my Queries?*” for a general overview of SQL operators, and check chapters *F.7* to *F.13* for details on **each** of the SQL operators.

If you want to know some good practices on writing your SQL Query code, you may click “*K.4 What Query design principles should I follow?*”.

If you want to know some good practices on handling **Null** in SQL Query code, you

may click “*K.5 Why and how should I carefully handle Nulls in my Queries?*”.

If you want to know how to write a few **useful Query models** (like doing partials and totals, or turning rows into columns), you may click “*K.6 What are some useful models of SQL code?*”.

If you want to know how to write **fast** Queries, you may click “*K.7 Why and how do I design a fast database and fast Queries?*”.

If you want to **debug** your Queries, you may click “***Part J. Debugging my SQL Queries***”.

## PART G. WRITING EXPRESSIONS

An **expression** is a mathematical **formula** composed of **variables** combined with **constants**, **operators** and/or **functions** that has **parts** of it enclosed between **parentheses**. If you want to know more, you may click to read “C.2.9 *What is an expression?*” and then return here (you return by simultaneously pressing the “**Alt**” and “**←**” keys).

MS-Access has **three different expression scopes**, and **within each scope** the rules to write **expressions** (variables, constants, operators, functions, etc.), are **not the same**. The **three MS-Access expression scopes** are:

- Expressions in **SQL code** (within user-defined Queries).
- Expressions in **VBA code** (within user-defined functions).
- Expressions in **Table “Design View”** (within record/field validation rules, field default values, calculated fields and “**Lookup**” property).

An expression is written by combining **constants**, **variables**, **operators**, **built-in functions** and **user-defined VBA functions**. **Parentheses** are used in the expression to indicate the desired **operator** evaluation order.

If you want to know the **main differences** between the writing rules (syntax) of the **three** expression scopes, you may click:

- “G.1 *What are the main differences between the three expression scopes?*”

If you want to know different aspects of writing expressions, while **explicitly** indicating the existing **differences** between the **three** expression scopes, you may click:

- “G.2 *How do I manage VBA data types and Table field types-sizes?*”
- “G.3 *What is the data type returned by an expression?*”
- “G.4 *How do I write a constant?*”
- “G.5 *How do I use value operators in an expression?*”
- “G.6 *How do I use functions in an expression?*”
- “G.7 *What is the evaluation order of an expression?*”
- “G.8 *How do I use an SQL operation in an expression?*”
- “G.9 *How are numeric-like values internally represented and processed?*”

### **G.1 What are the main differences between the three expression scopes?**

The three MS-Access expression scopes are:

- Expressions in **SQL code** (within user-defined Queries).
- Expressions in **VBA code** (within user-defined functions).
- Expressions in **Table “Design View”** (within record/field validation rules, field default values, calculated fields and “**Lookup**” property).

If you want to know, element by element, the main differences between writing expressions in these **three** expression scopes, you may click:

- “G.1.1 How available field names depend on expression scopes?”
- “G.1.2 How writing field names depends on expression scopes?”
- “G.1.3 How data types depend on expression scopes?”
- “G.1.4 How constants depend on expression scopes?”
- “G.1.5 How value operators depend on expression scopes?”
- “G.1.6 How non-aggregate built-in functions depend on expression scopes?”
- “G.1.7 How domain aggregate functions depend on expression scopes?”
- “G.1.8 How SQL aggregate functions depend on expression scopes?”
- “G.1.9 How user-defined functions depend on the expression scopes?”
- “G.1.10 How using SQL operations (“Subqueries”) depends on the expression scopes?”

### G.1.1 How available field names depend on expression scopes?

- **SQL expressions:**  
You **may** use any field name available in the SQL operation that encloses the expression. Remind that SQL expressions may have **specific restrictions** depending on the clause (“**SELECT**” **expression**, “**GROUP BY**” **expression**, ...) and the SQL operation (**Select**, **Join** or **Transform**). For example, in a “**GROUP BY**” **expression**, you may **not** use **output field names**. If you want to know more about this, you may click “F.11.1 Given an SQL clause, what are its expression’s elements?”.
- **VBA expressions:**  
You may **only** use field names for **assigning** the result of an SQL operation to an **VBA variable** (click K.9.3) and as arguments to **domain aggregate functions** (click G.6.2).
- **Table “Design View” expressions:**  
This depends on the specific element in **Table “Design View”**:
  - **Field default value:** cannot use **any** field name.
  - **Field validation rule:** can only use **the field name** to which the validation rule is associated.
  - **Calculated fields and record validation rule:** **all** the Table’s field names can be used.
  - “**Lookup**” property in Table/Form drop-down menus: **all** the Table’s field names can be used in the expressions. Notice that you can also write a Query, in which case you can access, through the Query, **all** fields from **all** Tables. If you want to know more about drop-down menus you may click “D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?”.

### G.1.2 How writing field names depends on expression scopes?

- **SQL expressions:**  
You **may** write **field names** enclosed in square brackets “[ ]”, and in some cases the square brackets are **mandatory** (click *D.2.5*). You **must** qualify (click *C.2.2*) **field names** in some cases (click *F.8*).
- **VBA expressions:**  
This only applies to assigning the result of an SQL operation to an VBA variable (click *G.8.5*). In this case, the rules for SQL code apply (click *F.6*).
- **Table “Design View” expressions:**  
You **must** write them enclosed in square brackets “[ ]”. You **do not** qualify them.

### G.1.3 How data types depend on expression scopes?

Data type aspects **do not** depend on expression scopes.

If you want to know more about data type aspects, you may click:

- [“G.2 How do I manage VBA data types and Table field types-sizes?”](#)
- [“G.3 What is the data type returned by an expression?”](#)

### G.1.4 How constants depend on expression scopes?

- **SQL expressions and Table “Design View” expressions:**  
You **may** write **True, Yes, On and False, No, Off** (all without quotes) as **Boolean** constants. You may also write **Null** (without quotes) as a constant. If you are using a foreign-language version, these constant names will be **translated**, as well as **month names** in dates.
- **VBA expressions:**  
You may **only** write **True** and **False** (both without quotes) as **Boolean** constants. You may also write **Null** (without quotes) as a constant.

If you want to know more about this, you may click [“G.4 How do I write a constant?”](#).

If you are using a foreign-language version of MS-Access, you may click [“L.8.12 How do I fix foreign-language issues of MS-Access?”](#).

### G.1.5 How value operators depend on expression scopes?

- **SQL expressions:**  
You **may** use **all** the value operators (click *G.5*).
- **VBA expressions:**  
You **may** use **all** the value operators (click *G.5*) except **“ALIKE”, “IS NULL”, “NOT IS NULL”, “IN” and “NOT IN”**.
- **Table “Design View” expressions:**  
You **may** use **all** the value operators (click *G.5*) except **“ALIKE”, “IS NULL”, and “NOT IS NULL”**.

To check which value operators you can use in **Table “Design View”** expressions, go to the **“Expression Builder”** box and click on **“Operators”**: this will show the available operators grouped by category. Remind that the **“Expression Builder”** box



is shown by clicking in the three-period “...” icon placed at the rightmost side of the properties “**Validation Rule**”, “**Default Value**” and “**Expression**” of the “**General**” Tab of field properties (the last one only for **Calculated** fields).

If you are using a foreign-language version, **some** (not all) **named operators** in this scope have **translated** names. For example, in the Spanish version the operators “**AND**”, “**OR**”, “**XOR**”, “**BETWEEN AND**” and “**NOT IN**” become “**Y**”, “**O**”, “**OE<sub>x</sub>**”, “**ENTRE Y**” and “**NoEs IN**”, while the operators, “**Mod**”, “**Imp**”, “**Eqv**”, “**IN**” and “**Like**” are **not translated** and stay with the same name.

The **separation character** between values in the lists of “**IN**” and “**NOT IN**” operators is **semicolon “;”** and **not comma “,”**. For example, you must write:

```
Quart IN ("Q1" ; "Q2" ; "Q3" ; "Q4")
```

If you want to know more about this, you may click “*G.5 How do I use value operators in an expression?*”.

If you are using a foreign-language version of MS-Access, you may click “*L.8.12 How do I fix foreign-language issues of MS-Access?*”.

### **G.1.6 How non-aggregate built-in functions depend on expression scopes?**

- **SQL expressions and VBA expressions:**  
You may use **almost** all the non-aggregate built-in functions in both scopes. Click **Part M** for a list of the available built-in functions, indicating for **each** of them if it can be used in SQL expressions and/or in VBA expressions.
- **Table “Design View” expressions:**  
The **separation character** between function **arguments** is **semicolon “;”**, and not **comma “,”**. For example, you must write:

```
Left([Quart] ; 1)
```

You may use **almost** the same non-aggregate built-in functions as in the other two scopes. To know which non-aggregate built-in functions you can use in **Table “Design View”** expressions, go to the “**Expression Builder**” box, click on “**functions**” and then click on “**Built-in functions**”: this will show the available built-in functions grouped by category. Remind that the “**Expression Builder**” box can be shown by clicking in the three-period “...” icon placed at the rightmost side of the properties “**Validation Rule**”, “**Default Value**” and “**Expression**” of the “**General**” Tab of field properties (the last one only for **Calculated** fields).

If you are using a foreign-language version, **most** built-in functions have **translated names** in this scope. For example, in the Spanish version the function names “**Left()**”, “**Right()**”, and “**Len()**” become “**Izq()**”, “**Der()**” and “**Longitud()**” in this scope.

If you want to know more about this, you may click “*G.6.1 How do I use non-aggregate built-in functions in an expression?*”.

If you are using a foreign-language version of MS-Access, you may click “*L.8.12 How do I fix foreign-language issues of MS-Access?*”.

### G.1.7 How domain aggregate functions depend on expression scopes?

- **SQL expressions and VBA expressions:**  
You **may** use **all** the **domain aggregate functions** in both scopes.
- **Table “Design View” expressions:**  
You **cannot** use **domain aggregate functions**.

If you want to know more about this, you may click “[G.6.3 How do I use SQL aggregate functions in an expression?](#)”.

### G.1.8 How SQL aggregate functions depend on expression scopes?

- **SQL expressions:**  
You **may** use **any** SQL aggregate function subject to the **restrictions** of the **specific** expression (“**SELECT**” **expression**, “**HAVING**” **expression**, ...) within the **specific** SQL operator (**Select-group\_by\_agg**, **Select-total\_agg** or **Transform**). For example, you **cannot** use them in “**WHERE**” expressions. If you want to know more about this, you may click “[F.11.1 Given an SQL clause, what are its expression’s elements?](#)”.
- **VBA expressions and Table “Design View” expressions:**  
You **cannot** use SQL aggregate functions.

If you want to know more about this, you may click “[F.7.18 What is an SQL aggregate function?](#)”.

### G.1.9 How user-defined functions depend on the expression scopes?

- **SQL expressions and VBA expressions:**  
You may use user-defined VBA functions.
- **Table “Design View” expressions:**  
You **cannot** use user-defined VBA functions.

If you want to know more about this, you may click “[K.9 How do I write my user-defined VBA functions and database Subroutines?](#)”.

### G.1.10 How using SQL operations (“Subqueries”) depends on the expression scopes?

- **SQL expressions:**  
You **may** use a **Select** operation, enclosed between parentheses, as a value (click [G.8](#)).
- **VBA expressions:**  
This only applies to assigning the result of an **SQL operation** to an VBA variable (click [G.8.5](#)). In this case, the rules for SQL code apply (click [F.6](#)).
- **Table “Design View” expressions:**  
You **cannot** use an **SQL operation** in field/record validation rule expressions, default field value expressions nor calculated field expressions. However, you can use an **SQL operation** in the “**Lookup**” property expressions of a Table/Form (click [D.11.1.1](#)).

If you want to know more about this, you may click “[G8 How do I use an SQL operation](#)”.

in an *expression*?”.

## G.2 How do I manage VBA data types and Table field types-sizes?

This chapter also answers the question:

- What are the data types used in expressions?

A **Table field type-size** is the **data type** (and the **field size** for **Number** data type) of **each field** that is defined in Table design (see “D.4 How do I configure a Table field data type and size?”).

The **VBA data types** are the data types defined in the programming language **VBA**.

The **data types** returned by **operators** and **functions** used in the **three expression scopes** are the **VBA data types**.

**VBA expressions only include VBA data types.**

**SQL expressions** and **Table “Design View” expressions** may include **Table fields** and **VBA function/operator results**, so they may include **both VBA data types and Table field types-sizes**. Therefore, there is a clear **interrelation** between **VBA data types** and **Table field types-sizes**.

Unfortunately, the **Table field types-sizes** and the **VBA data types** are **not exactly the same**. Luckily, **most Table field type-sizes** and **VBA data types** are **pairwise equivalent**, so you can **cross assign** them between Table fields and expression without any problem.

A value from a **Table field type-size** and a value from **its equivalent VBA data type** may be **cross assigned** between **expression variables** and **Table’s fields** as if they were of the same type. Therefore, you can assign a value from a VBA data type to a Table field that has its **equivalent** field type-size. Conversely, you can use a value from a Table field type-size as an operand of a VBA operator or as an argument of a VBA function that has its equivalent VBA data type. Even if both data types are **not** equivalent, you **can do** cross assignments, because MS-Access **automatically performs type conversions**, although doing this may cause some problems (click G.2.6).

If you want to know more about this, you may click:

- “G.2.1 What VBA data types vs. Table field types-sizes are equivalent?”
- “G.2.2 What VBA data types vs. Table field types-sizes are not equivalent?”
- “G.2.3 What is the result of combining different data/field types in expressions?”
- “G.2.4 How are data/field types grouped for easier reference?”
- “G.2.5 How do I force a value to belong to a specific data type?”
- “G.2.6 Why should I force a value into a specific data type?”

### G.2.1 What VBA data types vs. Table field types-sizes are equivalent?

The following table shows what **Table field types-sizes** (i.e., the field **type** plus the “**Field Size**” property for the **Number** field type) and what **VBA data types** are

**equivalent.** The table also indicates the details about storage space, range of values and precision:

Table		VBA		
Field Type	Field Size (property)	Data Type	Storage (Bytes)	Range of Values and Precision
Number	Byte	<i>Byte</i>	1	0 to 255 Precision: 3 significant decimal digits
	Integer	<i>Integer</i>	2	-32,768 $(-(2^{15}))$ to 32,767 $(2^{15}-1)$ Precision: 5 significant decimal digits
	Long Integer	<i>Long</i>	4	-2,147,483,648 $-(2^{31})$ to 2,147,483,647 $(2^{31}-1)$ Precision: 10 significant decimal digits
	Single	<i>Single</i>	4	Approx.: $\pm 1.17 \times 10^{-38}$ to $\pm 3.40 \times 10^{38}$ Precision: 6 to 9 significant decimal digits
	Double	<i>Double</i>	8	Approx.: $\pm 9.881 \times 10^{-324}$ to $\pm 1.797 \times 10^{308}$ Precision: 15 to 17 significant decimal digits
Yes/No		<i>Boolean</i>	2	<i>True</i> or <i>False</i> (numeric <b>-1</b> or <b>0</b> )
Currency		<i>Currency</i>	8	-922,337,203,685,477.5808 $-(2^{63})/10^4$ to 922,337,203,685,477.5807 $(2^{63}-1)/10^4$ Precision: 19 signif. dec. digits (4 decimals)
Date/Time		<i>Date</i>	8	January 1, 100 00:00:00 to December 31, 9999 23:59:59
Short Text		<i>String</i>	Variable	<b>Short Text</b> is 0 to 255 chars <b>String</b> is 0 to 65,400 chars
Long Text		<i>String</i>	Variable	0 to 65,400 chars
AutoNumber		<i>Long</i>	4	-2,147,483,648 to 2,147,483,648
OLE Object		<i>Object</i>	4	Object
Hyperlink		<i>String</i>	Variable	1 to ~65,400 chars
Large Number		<i>LongLong</i> (only 64b)	8	-9,223,372,036,854,775,808 $-(2^{63})$ to: 9,223,372,036,854,775,807 $(2^{63}-1)$ Precision: 19 significant decimal digits

Notice that **Large Number** Table field type is available in 32bit version of MS-Access while **LongLong** VBA data type is **not** available: the **LongLong** VBA data type is only available on a 64bit version of MS-Access.

For the case of **equivalent** data types, you may check the way VBA values are stored in Tables by checking the corresponding Table field types-sizes in “*D.4 How do I configure a Table field data type and size?*”.

You may check the official Microsoft site for complementary information on MS-Access data types:

<https://docs.microsoft.com/en-us/office/vba/access/concepts/error-codes/comparison-of-data-types>

### G.2.2 What VBA data types vs. Table field types-sizes are not equivalent?

The **Date/Time extended**, **Attachment** and **Number-Replication ID** Table field types-

sizes do not have an equivalent data type in VBA. The following table shows these Table field types-sizes with their characteristics:

Table		VBA		
Field Type	Field Size	Data Type	Storage (Bytes)	Range of Values
Number	Replication ID	N/A	16	Globally unique identifier required for replication
Date/Time extended		N/A	42	January 1, 1 00:00:00 to December 31, 9999 23:59:59.9999999
Attachment		N/A	N/A	N/A

In VBA you have other data types that have no equivalent in MS-Access SQL, like for example **arrays**. In VBA you can also define your own custom data types, that will usually have no equivalence to any MS-Access SQL field type.

It is important to mention the VBA *Variant* data type because it is frequently used for the arguments of user-defined functions, in order to prevent errors caused by data type mismatch when the function is invoked. The VBA *Variant* data type is something like a “**wildcard**” data type that will accept values from most other data types. The *Variant* data type is compatible with a value from almost any other data type.

### G.2.3 What is the result of combining different data/field types in expressions?

A **rough** summary of the result of combining values from different data/field types is:

- *True* is interpreted as **-1** and *False* is interpreted as **0**.
- **0** is interpreted as *False* and any other number is interpreted as *True*.
- Dates are interpreted as their numerical internal representation. Days are integers where **0** represents **30-December-1899**. Time are fractions of an integer, where 1/24 is one hour and so on. There are some tricky issues, so you may want to click “[D.4.5 What is the “Date/Time” field type?](#)” for a detailed explanation.
- The resulting data/field type from Boolean operators is always **Boolean**.
- The resulting **numeric-like** data/field type from an operation that combines two different **numeric-like** data/field types is the largest data/field type, unless it cannot hold the resulting value. In this case, the resulting data/field type is the smallest numeric data/field type that can hold the resulting value.
- Numeric-like aggregate functions (click [F.7.18.7](#)) work over **numeric-like** data/field types. They return a value of **numeric-like** data type.
- Calculation aggregate functions (click [F.7.18.7](#)) work over **string** or **numeric-like** data/field types. They return a value of the same data/field type as the expression of the function argument.
- “**First()**”, “**DFirst()**”, “**DLookup()**”, “**Last()**” and “**DLast()**” (click [F.7.18.7](#)) work over **any** data/field type and return an **integer**.
- “**Count()**”, “**DCount()**”, “**Count(\*)**” and “**DCount(“\*”)**” (click [F.7.18.7](#))

work over **any** data/field type, and return an integer.

If you want to know more about how Table field/size values are internally represented, and therefore, how will they be interpreted when combining them, you may check the different sections from “*D.4 How do I configure a Table field data type and size?*”. In particular, you may want to know how **Date/Time** values are internally represented clicking “*D.4.5 What is the “Date/Time” field type?*”.

If you want to know more about the range of values and the precision you may have with each data/field type, you may check the summary tables from:

- “*G.2.1 What VBA data types vs. Table field types-sizes are equivalent?*”
- “*G.2.2 What VBA data types vs. Table field types-sizes are not equivalent?*”

If you want to know more about how operators interpret values from different data/field types, you may click “*G.5 How do I use value operators in an expression?*”.

If you want to know more about what specific data type is returned by operators combining different data/field types, you may click “*G.3 What is the data type returned by an expression?*”.

If you want to know more about the data type of constants, you may click “*G.3.1 What is the data type of a constant?*”.

If you want to know more about how to force a value into a specific data type, you may click “*G.2.5 How do I force a value to belong to a specific data type?*”.

If you want to know details on decimal to binary conversion, you may click “*G.9.2 What is decimal/binary conversion?*”.

If you want to know details about rounding errors arising from decimal to binary conversion, you may click “*G.9.3 What are decimal/binary conversion rounding errors?*”.

## **G.2.4 How are data/field types grouped for easier reference?**

I have established the following groups of VBA data types and Table field type-sizes, according to their similar properties, to make it easier referencing them:

### **Integer data/field types**

I will jointly call “**integer**” data/field types to the VBA data types *Byte*, *Integer*, *Long*, and *LongLong*, plus their equivalent Table field types-sizes **Number-Byte**, **Number-Integer**, **Number-Long Integer** and **Large Number**.

Values from **different integer** data/field types can be combined in most operators functions and expressions.

### **Integer-like data/field types**

I will jointly call “**integer-like**” data/field types to the **integer** data/field types plus *Boolean* and *Yes/No* data/field types.

Therefore, the **integer-like** data/field types are the VBA data types *Boolean*, *Byte*, *Integer*, *Long*, and *LongLong*, plus their equivalent Table field types-sizes **Yes/No**, **Number-Byte**, **Number-Integer**, **Number-Long Integer** and **Large Number**.

Notice that *Boolean* and *Yes/No* values are represented as the integer “**-1**” for

**True/Yes/On** or **ticked** and the integer “0” for **False/No/Off** or **unticked**.

Notice also that number “0” is interpreted as **False**, and any non-zero number (even negative and/or fractional numbers) is interpreted as **True**, by most *Boolean* operators.

Values from **different integer-like** data/field types can be combined in most operators functions and expressions.

### **Fractional data/field types**

I will jointly call “**fractional**” data/field types to the VBA data types *Currency*, *Single* and *Double*, plus their equivalent Table field types-sizes **Currency**, **Number-Single** and **Number-Double**.

Values from **different fractional** data/field types can be combined in most operators functions and expressions.

### **Fractional-like data/field types**

I will jointly call “**fractional-like**” data/field types to the **fractional** data/field types plus *Date* and *Date/Time*.

Therefore, the **fractional-like** data/field types are the VBA data types *Currency*, *Single*, *Double* and *Date*, plus their equivalent Table field types-sizes **Currency**, **Number-Single**, **Number-Double** and **Date/Time**.

If you find strange that I included *Date* and *Date/Time* recall that *Date* and *Date/Time* values are **internally** represented using a **double** data/field type (click *D.4.5*). Notice also that the *Variant-Decimal* and *Decimal* data/field types are **not** included. This is so because the *Variant-Decimal* and *Decimal* data/field types have substantial drawbacks, and my advice is you do not use them (click *K.8*).

Values from **different fractional-like** data/field types can be combined in most operators functions and expressions.

### **Numeric data/field types**

I will jointly call “**numeric**” data/field types to the **integer** data/field types plus the **fractional** data/field types.

Therefore, the **numeric** data/field types are the VBA data types *Byte*, *Integer*, *Long*, *LongLong*, *Currency*, *Single* and *Double*, plus their equivalent Table field types-sizes **Number-Byte**, **Number-Integer**, **Number-Long Integer**, **Large Number**, **Currency**, **Number-Single** and **Number-Double**.

Values from **different numeric** data/field types can be combined in most operators functions and expressions.

### **Numeric-like data/field types**

I will jointly call “**numeric-like**” data/field types to the **integer-like** data/field types plus the **fractional-like** data/field types.

Therefore, the **numeric-like** data/field types are the VBA data types *Boolean*, *Byte*, *Integer*, *Long*, *LongLong*, *Currency*, *Single*, *Double* and *Date*, plus their equivalent field types-sizes: **Yes/No**, **Number-Byte**, **Number-Integer**, **Number-Long Integer**, **Large Number**, **Currency**, **Number-Single**, **Number-Double** and **Date/Time**.

Values from **different numeric-like** data/field types can be combined in most operators



functions and expressions.

### **Boolean data/field types**

I will jointly call “**Boolean**” data/field types to the VBA data type *Boolean* plus its equivalent Table field type **Yes/No**.

### **Currency data/field types**

I will jointly call “**currency**” data/field types to the VBA data type *Currency* plus its equivalent Table field type **Currency**.

### **String data/field types**

I will jointly call “**string**” data/field types to the VBA data type *String* plus its equivalent Table field type **Short Text**.

### **Date/time data/field types**

I will jointly call “**datetime**” data/field types to the VBA data type *Date* plus its equivalent Table field type **Date/Time**.

### **Long data/field types**

I will jointly call “**long**” data/field types to the VBA data type *Long* plus its equivalent Table field type-size **Number-Long**.

### **Single data/field types**

I will jointly call “**single**” data/field types to the VBA data type *Single* plus its equivalent Table field type-size **Number-Single**.

### **Double data/field types**

I will jointly call “**double**” data/field types to the VBA data type *Double* plus its equivalent Table field type-size **Number-Double**.

## **G.2.5 How do I force a value to belong to a specific data type?**

You **implicitly** force a value (either a **plain constant** or an **expression**) into a specific data type every time you:

- Type-in a value in a Table field.
- Assign a default value to a Table field.
- Write a constant or expression in your SQL code.
- Assign a value to a VBA variable (including a function argument when you invoke the function).

You can also **explicitly** force a value (either a plain constant or a more complex expression) into a specific data type using a “**type conversion**” function. You just enclose the value you want to force into a data type as the argument of the type conversion function corresponding to the data type that you want. You can use type conversion functions **over a whole** expression, or at as many places that you want **within** an expression. You can use the following built-in type conversion functions, all of which take only **one** argument of *Variant* data type:

- **CStr()**: returns the argument as a *String* data type.
- **CDate()**: returns the argument as a *Date* data type.

- **CBool()**: returns the argument as a *Boolean* data type.
- **CByte()**: returns the argument as a *Byte* data type.
- **CInt()**: returns the argument as an *Integer* data type.
- **CLng()**: returns the argument as a *Long* data type.
- **CCur()**: returns the argument as a *Currency* data type.
- **CSng()**: returns the argument as a *Single* data type.
- **CDbl()**: returns the argument as a *Double* data type.
- **CVar()**: returns the argument as a *Variant* data type.

The built-in type conversion function “**CLngLng()**” can only be used in 64bit versions of MS-Access and only in your VBA code. It can never be used in the SQL code, although you may write a user-defined VBA function that invokes it in VBA, and the user-defined function can be used in SQL.

Whenever you force a **value**, implicitly or explicitly, into a data type, the value is frequently **changed** because it must be **approximated** to the **closest value** that can be **represented** with the **binary format** of the corresponding data type. Some examples of values being changed are:

- **Cbyte(6.12335)** or **Cint(6.12335)**: return **6**, because *Byte* and *Integer* data types can only represent integer numbers.
- **CSng(6.12335)**: returns **6.12335014343262** because this is the closest value to 6. 12335 that can be represented with a *Single* data type.
- **CCur(6.12335)**: returns **6.1234** because it is rounded-half to even (banker’s rounding) to four decimal places, because the *Currency* data type can only have four decimal places.
- **CDbl(6.12335)**: returns exactly **6.12335** because a *Double* data type can store 6.12335 with sufficient precision. In this case the value is **not** changed.

### **G.2.6 Why should I force a value into a specific data type?**

In case of mixing different data types, it is much better that you do an explicit type conversion, so it is clear in your code that you that are mixing values with different data types. Also, when you do explicit type conversion you have more control on how it is done, and you avoid possible errors caused by MS-Access doing type conversion on his own.

You can perform explicit type conversion using the built-in type conversion functions from VBA: “**CBool()**”, “**CByte()**”, “**CCur()**”, “**CDate()**”, “**CDbl()**”, “**CInt()**”, “**CLng()**”, “**CSng()**”, “**CStr()**”, and “**CVar()**”. Each of these functions takes an argument from any data type and produces a result belonging to the specific data type of the function (function names are self-explanatory). If you want to know more about type conversion functions, you may click “[G.2.5 How do I force a value to belong to a specific data type?](#)”.

### **G.3 What is the data type returned by an expression?**

The data type returned by an expression is the one returned by the **last operator** or **function** that is **evaluated** in the expression. If the expression consists of **only one constant** or **variable**, then the data type is the one of the **constant** or **variable** (respectively).

Trusting your knowledge of the data type rules of MS-Access to determine what will be the data type returned by an expression is quite risky. If for some reason you need that an expression returns a **specific data type**, the best is that you enclose the expression in the corresponding data type conversion function (click [G.2.5](#)). Even if the expression is just a plain constant or variable, it is recommended to enclose it in a data type conversion function because in some cases (e.g., **Union** operations) MS-Access may for example interpret a **number** constant as a *String*.

You may also click:

- “[G.3.1 What is the data type of a constant?](#)”
- “[G.3.2 What is the data type returned by a non-arithmetic operator?](#)”
- “[G.3.3 What is the data type returned by an arithmetic operator?](#)”
- “[G.3.4 What is the data type returned by a function?](#)”

#### **G.3.1 What is the data type of a constant?**

Constants **True/Yes/On** or **ticked** and **False/No/Off** or **unticked** are VBA *Boolean* data type.

**Text string** constants are VBA *String* data type.

**Zero-time, zero-date and date-and-time** constants are VBA *Date* data type.

**Most** number constants do not belong to any specific data type<sup>65</sup>. Notice that the constants -5.6, -4, 0, 4 and 5.6 can be interpreted by each operator as a *Boolean*, *Byte*, *Integer*, *Long*, *LongLong*, *Currency*, *Single*, *Double* or *Date* data types.

If you want a number constant to surely belong to a specific data type, you can enclose it as an argument of the type conversion function “**CBool()**”, “**CByte()**”, “**CInt()**”, “**CLng()**”, “**CCur()**”, “**CSng()**” or “**CDBl()**” that forces the data type that you want.

If you want more information on type conversion functions, you may click “[G.2.5 How do I force a value to belong to a specific data type?](#)”.

#### **G.3.2 What is the data type returned by a non-arithmetic operator?**

**Logical**, **comparison** and **miscellaneous** operators always return a *Boolean* data type. They can also return **Null** or an **exception-value** (most frequently **type-error** “**#Type!**”).

**Text string** operators always return a *String* data type or **Null**. They can produce the exception value **type-error** if there is a data type mismatch. They can also produce **number-overflow** if the size of the string is larger than the processing range.

---

<sup>65</sup> On very few exception cases, if the specific value of a constant can only fit in one data type, then the constant can be univocally bound to that data type.

If you want more information about operators, you may click “[G.5 How do I use \*value operators\* in an \*expression\*?](#)”.

If you want to know more about **exception-values**, you may click “[J.15 What \*exception-value bugs\* can I get?](#)”.

### **G.3.3 What is the data type returned by an arithmetic operator?**

It depends on the specific operator and on the data types of its operands. Arithmetic operators can also return **Null** or an **exception-value**.

There are many combinations of data types and arithmetic operators, so instead of listing each case one by one I will summarize the way this works:

- The **division** “/” operator **always** returns a *Double*.
- The **integer division** “\” and **modulo** “Mod” operators **always** return a *Long* (or *LongLong*).
- The **change sign** operator “-” over an **integer-like** data type returns an *Integer* data type and over a **fractional** data type returns a **fractional** data type.
- The **remaining arithmetic operators** (^, \*, +, and -) return a *Long* (or *LongLong*) if **both** operands are either *Boolean* or an **integer-like** data type, and return a *Double* if one (or both) operands are either *Date* or a **fractional** data type.

If you want to know more about **operators**, you may click “[G.5 How do I use \*value operators\* in an \*expression\*?](#)”.

If you want to know more about **exception-values**, you may click “[J.15 What \*exception-value bugs\* can I get?](#)”.

### **G.3.4 What is the data type returned by a function?**

User-defined functions return the data type defined by the programmer.

Non-aggregate built-in functions return a data type specific for each function. If you want to know more about this, you may click “[Part M. List of built-in functions](#)”.

If you want to know the data-type returned by aggregate functions, you may click “[F.7.18.7 What is a summary and grouping of aggregate functions?](#)”.

## **G.4 How do I write a constant?**

A **constant** is a concrete fixed value. You can have constants of different VBA data types. The **constants** you can **write** in an **SQL expression**, in a **VBA expression**, and in a **Table “Design View” expression** have some **differences**. I will point out the differences between the said expression scopes when required. Constants are written in a very similar way to, **but with some differences**, typing-in a value into a Table/Form field in “**Datasheet View**” (click [E.2.2](#)).

Notice that in **Table “Design View” expressions** and in **VBA expression**, MS-Access will **change** the format of some constants that you type-in and **rewrite** them in its own standard format. For example, if in the VBA editor you type-in a zero-time constant corresponding to 30-December-1899 at 0:00:00, the VBA editor will **rewrite** it as “**#12:00:00 AM#**”, because this is numeric date **0** and numeric time **0**. As another

example, the VBA editor will **rewrite** every zero-time constant that you type-in into the format: “#month\_number-day\_number-4\_digit\_year\_number#”

You can use the **Null** constant in any expression by writing “Null” (without quotes).

Depending on the constant you want to write, you may click:

- “G.4.1 How do I write Boolean constants?”
- “G.4.2 How do I write String constants?”
- “G.4.3 How do I write Byte, Integer, Long Integer, LongLong, Currency, Single and Double constants?”
- “G.4.4 How do I write Date constants?”

I explain some additional restrictions when writing constants in VBA code in the section:

- “G.4.5 What are the restrictions when writing constants in VBA expressions?”

### G.4.1 How do I write Boolean constants?

**Boolean** constants are written differently depending on the scope of the expression:

- **SQL expressions** or **Table “Design View” expressions**  
You write the constants **True**, **Yes**, **On** or **False**, **No**, **Off** (all **without** quotes).
- **VBA expressions**  
You write the constants **True** and **False** (all without quotes). You **cannot** write the constants **Yes**, **No**, **On** nor **Off**.

If you are using a foreign-language version of MS-Access, in **Table “Design View” expressions** you have to write the **translated** constant names. For example, in the Spanish version you have to write **Verdadero**, **Sí**, **Activado** or **Falso**, **No**, **Desactivado** (always without quotes). You may want to click “L.8.12 How do I fix foreign-language issues of MS-Access?”.

### G.4.2 How do I write String constants?

A string of alphanumeric characters enclosed between **quotes**.

- **SQL expressions** or **Table “Design View” expressions**  
You can use either **double quotes** or **single quotes** to enclose the text string.
- **In VBA expressions**  
You can **only** use **double quotes** to enclose the text string.

You can include **double quote** characters in a string enclosed in **single quotes** and vice-versa. In case you want to include a **single** (or double) **quote** character within a string enclosed in **single** (or double, respectively) quotes, you only need to **duplicate the quote character** within the string.

Some examples of **SQL** and **Table “Design View”** string constants are: “ "Overlap dates" ”, “ 'US-Cities' ”, “ "Single ' in double" ”, “ 'Double " in single' ”, “ "Double "" in double" ”, “ 'Single " in single' ” (notice it is **two single** quotes and not **one double** quote).

Some examples of **VBA** string constants are: “ "Overlap dates" ”, “ "Engine\_Parts" ”,

“Single ' in double" ”, and “ "Delivery-Date" ”.

Remind that MS-Access is case insensitive, and therefore, for processing purposes the case will be ignored.

### **G.4.3 How do I write Byte, Integer, Long Integer, LongLong, Currency, Single and Double constants?**

For any of these data types, constants are written as positive and negative (except *Byte*) numbers in conventional decimal notation or in scientific notation. You **cannot** write a character for separation of thousands.

Some examples of integer number constants are: **123**, **-4564785**, **0**, **456E+12**, and **-15**.

Some examples of fractional number constants are: **4.0346**, **730**, **-325.234**, **235.67E26**, **.258E-15** and **-12.87E+25**.

If you are using a foreign-language version of MS-Access, in **Table “Design View” expressions** you have to write the **translated** decimal separator. For example, in the Spanish version you have to write “**3,5**” instead of “**3.5**”. You may want to click “*L.8.12 How do I fix foreign-language issues of MS-Access?*”.

### **G.4.4 How do I write Date constants?**

The way to write *Date* constants is **the same** as the way to type-in a value in a **Date/Time** Table field, but enclosing the value between “#” characters. If you want to know how to type-in a value in a **Date/Time** Table field, you may click:

- “*E.2.2.4 How do I type-in a value in a Date/Time field?*”

Like it happens when typing-in a value in a **Date/Time** Table field, a *Date* constant **always** represents **both** a **date** and a **time**. Even if you write a zero-time constant (i.e., a constant with only a date-part) or a zero-date constant (i.e., a constant with only a time-part), it represents **both** a **date-part** and a **time-part**. If you use a **zero-time constant** in an expression, it has the time-part **0:00:00**. If you use a **zero-date constant**, in an expression, it has the date-part **30-december-1899**.

Notice that in **Table “Design View” expressions** and in **VBA expressions**, MS-Access will **change** the format of *Date* constants that you type-in and **rewrite** them in its own standard format. For example, if in the VBA editor you type-in a zero-time constant corresponding to 30-December-1899 at 0:00:00, the VBA editor will **rewrite** it as “**#12:00:00 AM#**”, because this is numeric date **0** and numeric time **0**. As another example, the VBA editor will **rewrite** every zero-time constant you type-in into the format: “**#month\_number-day\_number-4\_digit\_year\_number#**”

### **G.4.5 What are the restrictions when writing constants in VBA expressions?**

In **VBA expressions** it is **not possible** to write some constants that you can write in **SQL expressions** and **Table “Design View” expressions**. The solution to this is building the constant value you want in a variable, writing operations over other constants.

For example, if you type-in in VBA the largest **Currency** constant:

```
922337203685477.5807
```

the VBA editor will rewrite it (this may be an MS-Access **bug**) as:

```
922337203685478#
```

The way you can get the actual value you want is by doing:

```
Curr_Var_A = 922337203685477#
Curr_Var_A = Curr_Var_A + 0.5807
```

and now the variable “Curr\_Var\_A” stores the value 922337203685477.5807 that you can use as you please in your VBA code.

As another example, if you type-in the following VBA constant:

```
9.781E-324
```

the VBA editor will rewrite it to:

```
9.88131291682493E-324
```

Because the decimal number “9.781E-324” cannot be represented in the binary format of a **Double** variable.

If you want to know more about data type formats and/or decimal to binary conversion, you may click:

- “G.2 How do I manage VBA data types and Table field types-sizes?”.
- “G.9 How are numeric-like values internally represented and processed?”

## G.5 How do I use value operators in an expression?

The value **operators** in your **SQL expressions**, **VBA expressions** and **Table “Design View” expressions** are **almost** the same, but they are **not** exactly the same. I will explain them together in this chapter, pointing out the slight differences between the **three** expression scopes when required.

Value operators are classified in the following groups:

- **Arithmetic** operators: “^”, “-” (change sign), “\*”, “/”, “\”, “Mod”, “+” and “-” (minus)  
You may click “G.5.1 What are the Arithmetic operators?”.
- **Text string** operators: “&” and “+”  
You may click “G.5.2 What are the Text string operators?”.
- **Comparison** operators: “=”, “<>”, “<”, “>”, “<=” and “>=”  
You may click “G.5.3 What are the Comparison operators?”.
- **Pattern** operators: “LIKE” and “ALIKE”  
You may click “G.5.4 What are the Pattern operators?”.
- **Logical (Boolean)** operators: “NOT”, “AND”, “OR”, “XOR”, “Eqv” and “Imp”  
You may click “G.5.5 What are the Logical (Boolean) operators?”.
- **Miscellaneous** operators: “IS NULL”, “IS NOT NULL”, “IN”, “NOT IN”,



“**BETWEEN AND**” and “**NOT BETWEEN AND**”.

Notice that **Miscellaneous** operators are **not** available in the three scopes. You may click:

- “G.5.6 What are the “*IS NULL*” and “*IS NOT NULL*” Miscellaneous operators?”.
- “G.5.7 What are the “*IN*” and “*NOT IN*” Miscellaneous operators?”.
- “G.5.8 What are the “*BETWEEN AND*” and “*NOT BETWEEN AND*” Miscellaneous operators?”.

If you are using a foreign-language version of MS-Access, you may click “L.8.12 How do I fix foreign-language issues of MS-Access?”.

### G.5.1 What are the Arithmetic operators?

Arithmetic operators are also called “mathematic” or “numeric” operators. You can use the following Arithmetic operators:

- Arithmetic operators with one operand: “-” (change sign).
- Arithmetic operators with two operands: “^” (exponential), “\*” (multiply), “/” (divide), “\” (integer division), “Mod” (modulo), “+” (plus) and “-” (minus).

Notice the **symbol** “-” used for the “change sign” operator and the “minus” operator is the same, but in spite of this they are **two different** operators.

The meaning of the above Arithmetic operators is obvious, except integer division and modulo:

- **Integer Division:**  $A \setminus B$  is the integer obtained from the expression.  

$$\text{Round\_towards\_zero}(\text{Round}(A) / \text{Round}(B))$$
- **Modulo:**  $A \text{ Mod } B$  is the integer **remaining amount** after dividing  $\text{Round}(A)$  by  $\text{Round}(B)$ .

Remind that the “**Round()**” built-in function performs **round-half to even** (banker’s rounding). If you want to know more about rounding types, you may click “J.11.20 How do I fix a Query making rounding errors?”.

#### **Arithmetic operators and data/field types**

Arithmetic operators work with operands of **numeric-like** data/field type, even if both operands are of a different data/field type. When both operands are of a different data/field type, then type conversion is applied **if required**. **True/Yes/On** or **ticked** is converted to the **integer-like** value “-1” while **False/No/Off** or **unticked** is converted to “0”. **Date** and **Date/Time** values are converted to the corresponding **Double** value (click D.4.5). **Integer-like** values may be converted to the **fractional** data type.

The data types returned by Arithmetic operators are:

- The **division** “/” operator **always** returns a **Double**.
- The **integer division** “\” and **modulo** “Mod” operators **always** return a **Long (or LongLong)**.
- The **change sign** operator “-” over an **integer-like** data type returns an **Integer** data type and over a **fractional-like** data type returns a **fractional-like** data type.

- The **remaining arithmetic operators** (^, \*, +, and -) return a *Long* (or *LongLong*) if **both** operands are either *Boolean* or an **integer-like** data type, and return a *Double* if one (or both) operands are *Date* or a **fractional** data type.

### Arithmetic operators and Null

If one (or both) operand(s) is/are **Null**, Arithmetic operators will return **Null**.

### Arithmetic operators and exception-values

If you use a *String* value as one, or both, operands the Arithmetic operators will return the exception-value **type-error** (shown as “#Error”). Notice though that using two text strings as the operands of the “+” operator will work well. The reason is that there are two “+” operators: one is the Arithmetic “+” operator (described in this subsection) and the other one is the “+” Text string operator (described in the next subsection G.5.2). Therefore, if you use two text string operands with the “+” operator, it will be interpreted as the “+” Text string operator, and it will work well.

An arithmetic operation that causes **overflow** returns the exception-value **number-overflow** (shown as “#Num!”). The operations that cause overflow are:

- **Adding, subtracting or multiplying** two *Byte*, *Integer* or *Long* values such that the result is out of range (positive or negative) of the *Long* data type.
- **Adding, subtracting or multiplying** a *Single* value with another *Single* value, or with any **integer-like** value, such that the result is out of range (positive or negative) of the *Double* data type.
- **Adding, subtracting or multiplying** a plain *Double* constant (i.e., a *Double* constant not enclosed in a “CDBl()” function) with any **integer-like** or *Single* value such that the result is out of range (positive or negative) of the *Double* data type.
- **Adding, subtracting or multiplying** a *Variant-Decimal* value with any **numeric** data type value such that the result is out of range (positive or negative) of the *Variant-Decimal* data type.
- Raising a number to a **power** such that the result is out of range (positive or negative) of the **output** data type.
- **Dividing 0 by 0**: “0/0”, “0\0” or “0 Mod 0”.

**Dividing non-zero by zero** with either of the operators “/”, “\” or “Mod” returns the exception-value **divide-by-zero** (shown as “#Div/0!”).

An exception-value as an operand of an Arithmetic operator returns **the same exception-value**.

## G.5.2 What are the Text string operators?

You can use the following Text string operators:

- Text string operators with two input operands: “&” and “+”

The “&” operator produces as output **one** text string composed of the first operand followed (concatenated) by the second operand. Leading or trailing spaces, other invisible characters or control characters are **not** ignored and **all** of them are

concatenated.

### Text string operators and data/field types

The “&” operator works with operands of **numeric-like** and **string** data/field types, even if both operands are of a different data/field type. When an operand is of **numeric-like** data/field types, its value is converted to a text strings as follows:

- In VBA code: **True** and **False** are converted to the **text strings** “**True**” and “**False**”
- In SQL and Table “**Design View**”: **True/Yes/On** or **ticked** is converted to the **text string** “**-1**” and **False/No/Off** or **unticked** is converted to the **text string** “**0**”
- Integer-like values and fractional values are converted to the **text string** that represents the number.
- **Date** and **Date/Time** values are converted to the **text string** that represents its date-part and/or its time-part.

The “+” **Text string operator** only works when **both its operands** are of **string** data/field types. Provided that both operands are of **string** data/field types, the “+” **Text string operator** returns the same value as the “&” operator.

The output data type of Text string operators is **String** (except when returning **Null** or an exception-value).

### Text string operators and Null

If **one** of the operands of the “&” operator is **Null** and the other is a valid text string, it will return the valid text string. If **both** operands of the “&” operator are **Null**, it will return **Null**.

If **one (or both)** operand(s) of the “+” operator is **Null**, it will return **Null**.

### Text-string operators and exception-values

Text-string operators **never return an exception-value** as long as none of its operands is an exception-value. Specifically, concatenating text strings resulting in a text string longer than 255 characters **does not** produce overflow. Actually, you can handle extremely long text strings (longer than 3,000 characters) in your expressions.

An exception-value as an operand of a Text-string operator returns **the same exception-value**.

## G.5.3 What are the Comparison operators?

You can use the following Comparison operators:

- Comparison operators with two operands: “=” (equal to), “<>” (different from), “<” (less than), “>” (greater than), “<=” (less than or equal to) and “>=” (greater than or equal to)

All Comparison operators **ignore trailing spaces** in their text string operands. This means that a given text string, and the same text string followed by any number of spaces produces **exactly the same result** in any comparison operator. **Leading spaces and trailing invisible characters other than spaces** (e.g., tabs or new-line characters) **are**

**not ignored** and will be taken into account by all the Comparison operators.

Remind that MS-Access is case insensitive, and therefore text strings are compared without taking into account the case. For example, the expression:

```
"PaRiS" = "pArIs"
```

is evaluated to **True**.

### Comparison operators and data/field types

Comparison operators work with either **both** operands of **numeric-like** data/field type or **both** operands of **string** data/field type.

When the two operands are of a **different numeric-like** data/field type, type conversion will take place **if required**. The value **True/Yes/On** or **ticked** is converted to the **integer-like** value “-1” while **False/No/Off** or **unticked** is converted to **integer-like** value “0”. **Date** and **Date/Time** values are converted to the corresponding **Double** value (click *D.4.5*). **Integer-like** values may be converted to **fractional** values.

The output data type of Comparison operators is **Boolean** (except when returning **Null** or an exception-value).

### Comparison operators and Null

If one (or both) operand(s) is/are **Null**, all Comparison operators return **Null**.

### Comparison operators and exception-values

If one operand is a **string** and the other operand is of **other** data type, the result is the exception-value **type-error** (shown as “#Error”).

An exception-value as an operand of a Comparison operator returns **the same exception-value**.

## G.5.4 What are the Pattern operators?

The “**ALIKE**” operator **does not exist** in **VBA scope**, nor in **Table “Design View”** scope, but you can use the “**LIKE**” operator in the three scopes.

The “**LIKE**” operator is much more flexible than the “**ALIKE**” operator. Using both on my view only introduces confusion. I advise you only use the “**LIKE**” operator, and for this reason I will not explain the “**ALIKE**” operator.

The “**LIKE**” operator returns **True** if its first operand **matches** the pattern indicated in its second operand.

The second operand of the “**LIKE**” operator is a text **string** that indicates the **pattern** to be matched. The following table lists the semantics of the different pattern characters that you can use to compose a **pattern** text string:

Kind of match	Pattern	Match (returns True)	No match (returns False)
Multiple characters	a*a	aa, aBa, aBBBa	aBC
	ab	abc, AABb, Xab	aZb, bac

Kind of match	Pattern	Match (returns True)	No match (returns False)
Special character	a[*]a	a*a	Aaa
Multiple characters	ab*	abcdefg, abc	cab, aab
Single character	a?a	aaa, a3a, aBa	aBBBa
Single digit	a#a	a0a, a1a, a2a	aaa, a10a
Range of characters	[a-z]	f, p, j	2, &
Outside a range	[!a-z]	9, &, %	b, a
Not a digit	[!0-9]	A, a, &, ~	0, 1, 9
Combined	a[!b-m]#	An9, az0, a99	abc, aj0

### Pattern operators and data/field types

The “**LIKE**” operator works with operands of any **numeric-like** and **string** data/field type, even if the two operands are of different data/field types. If the second operand is not **string**, its value will be converted to a text string value.

The output data type of the “**LIKE**” operator is *Boolean* (except when returning **Null** or an exception-value).

### Pattern operators and Null

If one (or both) operand(s) is/are **Null**, the “**LIKE**” operator returns **Null**.

### Pattern operators and exception-values

The “**LIKE**” operator **never returns an exception-value**, as long as none of its operands is an exception-value.

An exception-value as an operand of a the “**LIKE**” operator returns **the same exception-value**.

## G.5.5 What are the Logical (*Boolean*) operators?

You can use the following Logical operators:

- Logical operator with one operand: “**NOT**” (change *Boolean* value).
- Logical operators with two operands: “**AND**” (logical “and”), “**OR**” (logical “or”), “**XOR**” (exclusive “or”), “**Eqv**” (logical equality) and “**Imp**” (logical implication).

If you are using a foreign-language version, in **Table “Design View” expressions** the “**NOT**”, “**AND**”, “**OR**” and “**XOR**” operators may be **translated**. For example, in the Spanish version they become “**NoEs**”, “**Y**”, “**O**” and “**OEx**”. If you want to know more about this, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

### Logical operators and data/field types

The operands of the Logical operators can be of any **numeric-like** or **string** data/field type, even if the two operands are of different data/field types.

The numeric value “**0**” is interpreted as **False**, and **any other** numeric value is interpreted as **True**. The **datetime** value “**30-dec-1899 0:00:00**” (which is numeric zero,

click *D.4.5*) is interpreted as **False**, and any other **datetime** value is interpreted as **True**. The text string “0” is interpreted as **False**, and any other text string (including the **zero-length** string) is interpreted as **True**.

The output data type of Logical operators is **Boolean** (except when returning **Null** or an exception-value).

### Logical operators and Null

If one (or both) operand(s) is/are **Null**, Logical operators returns **Null**, except in the following cases:

- “False **AND** Null” and “Null **AND** False” return “False”
- “True **OR** Null” and “Null **OR** True” return “True”
- “False **Imp** Null” returns “True”
- “Null **Imp** True” returns “True”

Notice that the results in the cases above is what you would expect if **Null** is replaced by “**unknown**” or by a variable **X**.

### Logical operators and exception-values

Logical operators **do not return an exception-value**, unless one (or both) of its operands is an exception-value.

An exception-value as an operand of a Logical operator returns **the same exception-value**.

## G.5.6 What are the “IS NULL” and “IS NOT NULL” Miscellaneous operators?

Operators “**IS NULL**” and “**IS NOT NULL**” **do not exist in VBA scope**, nor in **Table “Design View” scope** (but you can use the “**IsNull()**” function).

“**IS NULL**” returns **True** if its operand is **Null** and **otherwise** returns **False**.

“**IS NOT NULL**” returns **False** if its operand is **Null** and **otherwise** returns **True**.

Notice that for the operators “**IS NULL**” and “**IS NOT NULL**”, the operand goes **before** the operator (i.e., the operand is to the **left** of the operator). This on my view makes their readability somehow poor.

### Operators “IS NULL” and “IS NOT NULL” and data/field types

Operators “**IS NULL**” and “**IS NOT NULL**” work with and operand of any **numeric-like** or **string** data/field type.

The output data type of “**IS NULL**” and “**IS NOT NULL**” operators is **Boolean** (except when returning an exception-value).

### Operators “IS NULL” and “IS NOT NULL” and Null

Operators “**IS NULL**” and “**IS NOT NULL**” **never return Null**.

### Operators “IS NULL” and “IS NOT NULL” and exception-values

Operators “**IS NULL**” and “**IS NOT NULL**” **do not return an exception-value**, unless its operand is an exception-value.

An exception-value as an operand of “**IS NULL**” or “**IS NOT NULL**” operators returns **the same exception-value**.

### **G.5.7 What are the “IN” and “NOT IN” Miscellaneous operators?**

Operators “**IN**” and “**NOT IN**” **do not exist** in VBA scope.

Operators “**IN**” and “**NOT IN**” **exist** in Table “**Design View**” scope, but in this scope the **separator character** between values in the list is **semicolon “;”** and is **not comma “,”**.

Operator “**A IN S\_list**” returns **True** if the searched value “**A**” is one of the values in the searched list “**S\_list**” and returns **False** otherwise. Operator “**A NOT IN S\_list**” returns **the opposite**. The list of values in the searched list “**S\_list**” has to be enclosed in parentheses and individual values are separated with a comma “**,**”. I want to highlight that text strings are compared without taking into account the case (i.e., case insensitive). For example, the following expression:

```
"New Dehli" IN ("new dehli", "other_value")
```

is evaluated to **True**.

If you are using a foreign-language version, in Table “**Design View**” expressions the “**IN**” and “**NOT IN**” operators may be **translated**. For example, in the Spanish version “**NOT IN**” becomes “**NoEs IN**”. If you want to know more about this, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

#### **Operators “IN” and “NOT IN” and data/field types**

The following combinations of operand data/field types work:

- If searched value is a **numeric-like** data/field type, then, the searched list “**S\_list**” may contain values from any **numeric-like** data/field type, even if they are different.
- If searched value is a **string**, then, the searched list “**S\_list**” may contain values from **string**, **Boolean** and **datetime** data/field types, even if they are different.
- If searched value is a **datetime**, then, the searched list “**S\_list**” may contain values from any **numeric-like** and **string** data/field types, even if they are different.

If an “**IN**” or “**NOT IN**” operator does not comply with the three data/field type rules above, the Query will **crash**, with the **error** message:

```
“Data type mismatch in criteria expression.”
```

As an exception, if the searched value is found (left to right) in the searched list **before** having reached the value with the data type mismatch, then the operator will **not crash**, and it will return the corresponding **Boolean** value.

The output data type of “**IN**” and “**NOT IN**” operators is **Boolean** (except when returning **Null**).

#### **Operators “IN” and “NOT IN” and Null**

If the **searched value** is **Null**, the operators will return **Null**.

If the **searched list** “**S\_list**” contains one or more **Null** they will be ignored, and the



operator will work normally.

### Operators “IN” and “NOT IN” and exception-values

The “IN” and “NOT IN” operators **never return** an exception-value.

An **exception-value** as an **operand** of operators “IN” or “NOT IN” will cause the following:

- If the exception-value is the **searched value**, the Query will **crash**.
- If the exception-value is one of the values in the searched list, and also, the searched value is **not** to its left in the list (i.e., the exception-value is reached before having found the searched value), the Query will **crash**.
- Otherwise, a correct value will be returned by the “IN” or “NOT IN” operators.

### G.5.8 What are the “BETWEEN AND” and “NOT BETWEEN AND” Miscellaneous operators?

Operators “BETWEEN AND” and “NOT BETWEEN AND” **do not exist** in VBA scope.

Operator “B BETWEEN A AND C” returns **True** if “A” is less than or equal to “B”, and also, “B” is less than or equal to “C”. This is equivalent to:

```
(( A <= B ) AND ( B <= C ))
```

Operator “X NOT BETWEEN A AND C” returns **True** if “X” is less than “B”, and/or, “C” is less than “X”. This is equivalent to:

```
(( X < B ) OR ( C < X ))
```

Notice that these operators are **conceptually comparison** operators, but they are most frequently categorized as Miscellaneous.

If you are using a foreign-language version, in Table “Design View” expressions the “BETWEEN AND” and “NOT BETWEEN AND” operators may be **translated**. For example, in the Spanish version they become “Entre Y” and “NoEs Entre Y”. If you want to know more about this, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

### Operators “BETWEEN AND” and “NOT BETWEEN AND” and data/field types

Operators “BETWEEN AND” and “NOT BETWEEN AND” work with operands of any **numeric-like** or **string** data/field type, even if its three operands are of different data/field types, with **one** exception. The exception is if you combine **string** operand(s) with **integer-like** and/or **fractional** operand(s): this will return the exception-value **type-error** (shown as “#Error”).

If you combine **datetime** operand(s) with **string** operand(s), the **datetime** operand(s) is(are) converted to a **string**. If you combine **datetime** operand(s) with **integer-like** and/or **fractional** operand(s), the **datetime** operand(s) is(are) interpreted as its numeric representation (click [D.4.5](#)).

The output data type of “IS NULL” and “IS NOT NULL” operators is **Boolean**

(except when returning **Null** or an exception-value).

### Operators “**BETWEEN AND**” and “**NOT BETWEEN AND**” and Null

If one (or more) operand(s) is/are **Null**, the operators “**BETWEEN AND**” and “**NOT BETWEEN AND**” return **Null**.

### Operators “**BETWEEN AND**” and “**NOT BETWEEN AND**” and exception-values

Operators “**BETWEEN AND**” and “**NOT BETWEEN AND**” return the exception-value **type-error** (shown as “**#Error**”) if you combine **string** operand(s) with **integer-like** and/or **fractional** operand(s).

An exception-value as an operand of “**BETWEEN AND**” or “**NOT BETWEEN AND**” operators returns **the same exception-value**.

## G.6 How do I use functions in an expression?

A function is a **named** mathematical computation that takes some variables as input and produces one single output value as a result. Each time you invoke the function with some concrete input values, the function is computed, and it returns one single concrete output value. The input variables of a function are usually called “arguments”.

Functions (with **very few** exceptions) are **deterministic** and **do not** have memory. This means that **every time** you **run** a **function** over the **same arguments**, you get the **same result**.

Most functions belong to **one** data type, corresponding to the data type of **the value that the function returns**. For example, **Boolean** functions are the ones that produce a **Boolean** value, and **Integer** functions are the ones that produce an **Integer** value. Some examples of functions, and their data type are:

```
Round(Floating_variable, 0)           (returns an Integer)
Is_Null(Input_Variable)              (returns a Boolean)
Iif(Age>0, (Height/Age)^2, "Error")  (returns a Variant)
```

The arguments of a function may belong to different data types. You may check the three example functions above and see that they contain arguments of different data types. Most functions have a **fixed** number of arguments. Most functions have arguments **each** of them belonging to a **specific** data type.

Notice however that there are some functions that return a **Variant** (click *G.2.2*) and/or that have one or more arguments of **Variant** data type. It is also possible to have functions with a **variable** number of arguments.

The way to use (also called “invoke” or “instantiate”) a function is by writing its name followed by its arguments separated by **commas** (except in **Table “Design View”**, that uses **semicolons**) and enclosing **all** the arguments between parentheses. Some examples are:

```
Round(10.45, 2)
Is_Null("Ford_Mustang")
Overlap_dates(#3/1/1980#, #6/23/1982#, #1/1/1981#, #12/31/1981#)
```

Depending on the type of function you want to use, you may click:

- “*G.6.1 How do I use non-aggregate built-in functions in an expression?*”

- “G.6.2 How do I use domain aggregate functions in an expression?”
- “G.6.3 How do I use SQL aggregate functions in an expression?”
- “G.6.4 How do I use user-defined VBA functions in an expression?”

## G.6.1 How do I use non-aggregate built-in functions in an expression?

Non-aggregate built-in functions can be used in the following scopes:

- **SQL expressions and VBA expressions:**  
You **may** use **almost** all the non-aggregate built-in functions in both scopes. Click *Part M* for a list of the available built-in functions, indicating for **each** of them if it can be used in SQL expressions and/or VBA expressions.
- **Table “Design View” expressions:**  
The **separation character** between function **arguments** is **semicolon “;”**, and not **comma “,”**. For example, you must write:

```
Left([Quart] ; 1)
```

You may use **almost** the same non-aggregate built-in functions as in the other two scopes. To know which non-aggregate built-in functions you can use in **Table “Design View”** expressions, go to the “**Expression Builder**” box, click on “**functions**” and then click on “**Built-in functions**”: this will show the available built-in functions grouped by category. Remind that the “**Expression Builder**” box can be shown by clicking in the three-period “**...**” icon placed at the rightmost side of the properties “**Validation Rule**”, “**Default Value**” and “**Expression**” of the “**General**” Tab of field properties (the last one only for **Calculated** fields).

If you are using a foreign-language version, **most** built-in functions have **translated names** in this scope. For example, in the Spanish version the function names “**Left()**”, “**Right()**”, and “**Len()**” become “**Izq()**”, “**Der()**” and “**Longitud()**” in this scope.

Among all the built-in functions, I want to mention here the following ones that I believe are used very frequently:

- [Iif\(Boolean exp, output exp 1, output exp 2\)](#)  
If its argument **Boolean** expression “**Boolean\_exp**” evaluates to **True**, “**Iif()**” returns the value of the expression “**output\_exp\_1**”, and otherwise it returns the value of the expression “**output\_exp\_2**”. Notice that if “**Boolean\_exp**” is evaluated to **Null**, the value of “**output\_exp\_2**” is returned.

**Important remark:** the output expression that is not returned is **not evaluated**, and therefore, in case it is erroneous, the function **will not crash**. However, the “**Switch()**”, “**Choose()**” and “**Nz()**” functions **evaluate all** the expressions in their arguments, regardless of which one is returned. In this sense, the “**Iif()**” function is more efficient and in particular, **is much safer** than the “**Switch()**”, “**Choose()**” or “**Nz()**” functions.

**Important warning!** if **both** output expressions are of a **numeric-like** data type, the returned result will be of **numeric-like** data/field type. However, if **either** (or both) output expressions is a **string**, the returned result will be a **String**.

- [Switch\( Bexp-1, Oexp-1 \[, Bexp-2, Oexp-2 \] ... \[, Bexp-n, Oexp-n \] \)](#)

Its arguments consist of a list of **pairs**, each pair being one *Boolean* expression plus one output expression. The function returns the result of the output expression associated to the **leftmost Boolean** expression that evaluates to *True*.

If one or more *Boolean* expressions evaluates to **Null** the function works normally, and the only effect is that the associated output expression is not returned.

If **none** of the *Boolean* expression evaluates to *True*, then **Null** is returned.

**Important Warning!** the “Switch()” function **evaluates all** the *Boolean* expression and **all** the expressions in its arguments, regardless of which one is returned. This implies that if **any** of the expressions **crashes** (e.g., 0/0, 4/0, ...), even if that expression is to the right of the returned value, the “Switch()” function **will also crash**. This **does not happen** with the “If()” function, that in this sense is much safer.

**Important warning!** if **all** output expressions are of a **numeric-like** data/field type, the returned result will be of **numeric-like** data type. However, if **one** (or more) output expressions is a **string**, the returned result will be a *String*.

- [Choose\(index\\_exp, output\\_exp\\_1, \[output\\_exp\\_2\], ...\)](#)

It returns the result of one of the expressions output\_exp\_n based on the resulting value of its first argument “index\_exp” expression. If “index\_exp” evaluates to 1, it returns the result of “output\_exp\_1”; if “index\_exp” evaluates to 2, it returns the result of “output\_exp\_2”, and so on. If “index\_exp” evaluates to a fractional value, it is rounded **down**.

If the rounded result of “index\_exp” is less than 1 or greater than the number of “output\_exp” arguments, then **Null** is returned.

**Important warning!** the “Choose()” function **evaluates all** the expressions in its arguments, regardless of which one is returned. This implies that if **any** of the expressions **crashes** (e.g., 0/0, 4/0, ...), even if that expression is not the returned value, the “Choose()” function **will also crash**. This **does not happen** with the If() function, that in this sense is much safer.

**Important warning!** if **all** the output expressions are of a **numeric-like** data/field type, the returned result will be of **numeric** data type. However, if **one** (or more) output expressions is a **string**, the returned result will be a *String*.

- [IsNull\(input\\_exp\)](#)

Returns **True** if its argument expression “input\_exp” evaluates to **Null**, and otherwise returns **False**.

- [Nz\(input\\_output\\_exp, output\\_exp\\_if\\_Null\)](#)

If its argument expression “input\_output\_exp” is evaluated to a **non-Null value**, it returns that **value**, and otherwise returns the result of “output\_exp\_if\_Null”.

**Important warning!** even if the first argument evaluates to a **non-Null value**, the expression in the second argument **will be evaluated**. For example, the following function invocation will crash: “Nz(5, 0/0)”. This **does not happen** with the “If()” function, that in this sense is much safer.

**Important warning!** when used in an SQL operation, “Nz()” **always** returns a *String*, regardless of the data type of its arguments. If you want it to return a

number or a date, enclose “Nz()” in the corresponding type-conversion function (e.g., “Cdbl(Nz(X, 0))”).

- [Format\(input\\_exp \[, format\\_string \] \[, firstdayofweek \] \[, firstweekofyear \] \)](#)  
Returns the value of the expression input\_exp formatted as a **text string** using the format indicated by the optional argument format\_string. The other two optional arguments are related to options in **Date** formatting. If you want to know more about the different formatting you can get depending on the value of format\_string, you may click “[\*H.6.2 How do I configure custom column formatting in a Table/Query/Form?\*”](#).”
- [Date\(\)](#)  
Returns the current date with zero-time. It is frequently used as the default value of Table fields with **Date/Time** field type.
- [DateSerial\(year, month, day\)](#)  
With **integer-like** arguments, it returns the date (with zero-time) corresponding to the **year**, **month** and **day** in its arguments.

Notice that the **month** argument can be zero, negative, or larger than 12. Month 0 is December of previous year. Month -1 is November of previous year, and so on modulus 12. Month 13 is January of next year. Month 14 is February of next year, and so on modulus 12.

Notice that the **day** argument can be zero, negative, or larger than the month length. Day 0 is last day of previous month (and previous year, if applicable). Day -1 is last but one day of previous month (and previous year, if applicable). Day -364 is first day of month, one year earlier (assuming it is not a leap year). Day 366 is first day of month, one year later (assuming it is not a leap year). The same applies if you use day number smaller than -364 or larger than 366. Month durations are always computed correctly, taking into account the month duration and the leap years.

The **day** or **month** arguments can also be **fractional**. In this case, the function will round them to an **integer-like** value using round-half to even. If you want to know more about rounding types, you may click “[\*J.11.20 How do I fix a Query making rounding errors?”\*”.](#)

If you want more detail about any of these functions, you can click on the function title in the bullets above and your navigator will open on the corresponding function description in the official Microsoft support web site.<sup>66</sup>

## G.6.2 How do I use domain aggregate functions in an expression?

Domain aggregate functions are one of the groups of built-in functions (click **Part M**). They are somehow special because they can **extract** aggregate **data from** your **database Tables**. Domain aggregate functions can be used in the following scopes:

- **SQL expressions and VBA expressions:**  
You **may** use **all** the domain aggregate functions in both scopes.
- **Table “Design View” expressions:**  
You **cannot** use domain aggregate functions.

---

<sup>66</sup> Notice that the web pages in the Microsoft support site may have changed since this book was released, and these links may not work.

A **domain aggregate function** returns the same result as a **Select-total\_aggreg with its dual SQL aggregate function** (click *F.7.18.7*) over a single **Table/Query**. A simple example of the “**DCount()**” domain aggregate function is:

```
DCount("City", "T_Capital_Temps", "Temp_Max > 4")
```

This domain aggregate function returns **exactly the same result** as the following **Select-total\_aggreg**:

```
SELECT Count(City) FROM T_Capital_Temps WHERE Temp_Max > 4
```

**Domain aggregate functions** can be **directly** used in your user-defined VBA functions and subroutines, in your macros, in the expressions within your SQL code and in your calculated controls. You can even use **domain aggregate functions** in the expressions that are the arguments of a **domain aggregate function**, although using the quotes correctly gets difficult.

Notice that **domain aggregate functions** have the **same basic functionality** as **SQL aggregate functions** (click *F.7.18* and *G.6.3*). However, they also have **relevant differences**. One such difference is that **SQL aggregate functions** can **only** be used in **some** expressions within a **Select-group\_by\_aggreg**, within a **Select-total\_aggreg** or within a **Transform** operation. Conversely, **domain aggregate functions** can be **directly** used in your user-defined VBA functions and subroutines, in your macros, in **any** expression within your SQL code and in your calculated controls.

The correct way to write (syntax) a **domain aggregate function** is:

```
DAgg_func("exp(In-flds)", "Table-name or Query-name" [, "Where-bool-exp()"])
```

The **first** argument is a text string containing an expression over the fields of the input record-list, this is, over the fields of the Table or Query in the second argument. This expression **cannot** contain a **domain aggregate function**.

The **second** argument is a text string containing a Table name or a Query name over which the **domain aggregate function** will be applied. In case you use a Query name, that Query **cannot** have **parameters**.

The **third** argument is enclosed in square brackets to indicate that it is optional. If it exists, the third argument is a text string containing a **Boolean** expression that the records of the Table or Query must match (i.e., produce **True**) in order to serve as input to the domain aggregate function. If this third argument does not exist, the input record-list is the complete record-list of the Table or Query in the second argument.

The **domain aggregate function** above returns **exactly the same result** as the following **Select** operation:

```
SELECT Agg_func(exp(In-flds))
FROM Table-name or Query-name
[WHERE Where-bool-exp()]
```

where “**Agg\_func()**” represents the corresponding **dual SQL aggregate function** to the generic domain aggregate function “**DAgg\_func()**” above.

In case you need to use text strings in the expression “**exp()**” and/or in the **Boolean** expression “**Where-bool-exp()**”, you **must** enclose each text string in single quotes (and **not** in double quotes). For example:

```
DCount("City & 'Temp'", "T_Capital_Temps", "City <> 'Chicago'")
```

For **each SQL** aggregate function (click *F.7.18*) there is one **dual domain** aggregate function that has its same name prefixed by “**D**”. For example, the “**Count()**” SQL aggregate function is **dual** to the “**DCount()**” domain aggregate function, and the “**Avg()**” SQL aggregate function is **dual** to the “**DAvg()**” domain aggregate function. The two **dual** aggregate functions in each **pair** have the **same basic functionality**.

Therefore, MS-Access offers the following **domain aggregate functions**:

- **DCount**(“\*”, “Table-name or Query-name” [, “Bool-exp()”])  
Click *F.7.18.1* for a description of its functionality.
- **DCount**(“exp()”, “Table-name or Query-name” [, “Bool-exp()”])  
Click *F.7.18.2* for a description of its functionality.
- **DFirst, DLast**(“exp()”, “Table-name or Query-name” [, “Bool-exp()”])  
Click *F.7.18.3* for a description of their functionality.
- **DMin, DMax**(“exp()”, “Table-name or Query-name” [, “Bool-exp()”])  
Click *F.7.18.4* for a description of their functionality.
- **DSum, DAvg**(“exp()”, “Table-name or Query-name” [, “Bool-exp()”])  
Click *F.7.18.5* for a description of their functionality.
- **DStDev, DStDevP, DVar, DVarP**(“exp()”, “Table-name or Query-name” [, “Bool-exp()”])  
Click *F.7.18.6* for a description of their functionality.

In addition to providing the previous **domain aggregate functions** (one for each **dual SQL** aggregate function), MS-Access provides the following **additional domain** aggregate function:

**DLookup**(“exp()”, “Table-name or Query-name” [, “Bool-exp()”])

The function “**DLookup()**” returns the result of the expression “exp()” computed over the **first** record from the Table/Query. If the optional third argument exists, then the result will be computed over the **first** record among the ones that produce **True** in “Bool-exp()”.

What I just said **seems** to be the same functionality as the “**DFirst()**” domain aggregate function, but there is a **very** important difference between both functions. “**DFirst()**” selects the **first** record according to the **internal** record ordering of MS-Access, which means returning one **arbitrary** record. However, “**DLookup()**” returns the **first** record according to the “**ORDER BY**” record ordering of its input record-list!

Aside from this **important** difference, the functionality of “**DLookup()**” is similar to the one of “**DFirst()**”, which in turn, is similar to the one of “**First()**” (click *F.7.18.3*).

### G.6.3 How do I use SQL aggregate functions in an expression?

SQL aggregate functions can be used in the following scopes:

- **SQL expressions**:  
You **may** use **any** SQL aggregate function subject to the **restrictions** of the **specific** expression (“**SELECT**” **expression**, “**HAVING**” **expression**, ...) within the **specific** SQL operator (**Select-group\_by\_aggreg**, **Select-total\_aggreg** or **Transform**). For



example, you **cannot** use them in “**WHERE**” expressions. If you want to know more about this, you may click “*F.11.1 Given an SQL clause, what are its expression’s elements?*”.

- **VBA expressions and Table “Design View” expressions:**  
You **cannot** use SQL aggregate functions.

If you want to know more about this, you may click “*F.7.18 What is an SQL aggregate function?*”.

### **G.6.4 How do I use user-defined VBA functions in an expression?**

User-defined VBA functions can be used in the following scopes:

- **SQL expressions and VBA expressions:**  
You may use user-defined VBA functions.
- **Table “Design View” expressions:**  
You **cannot** use user-defined VBA functions.

If you want to know more about this, you may click “*K.9 How do I write my user-defined VBA functions and database Subroutines?*”.

### **G.7 What is the evaluation order of an expression?**

An expression produces **different results** depending on the **evaluation order** (also called “computation” order) of the **operators** within the expression. For example, the expression “ $3*4+2$ ” produces **14** but it produces **18** if you add parentheses to make it “ $3*(4+2)$ ”.

The evaluation order of an expression is determined by the following rules:

1. Expressions between **parentheses** are **always** evaluated first. If there are nested parentheses, the **innermost** expressions between parentheses are evaluated first.
2. In an expression **without** parentheses the operators with **highest precedence** are evaluated first.
3. In an expression **without** parentheses, and with operators of the **same precedence**, the expression is evaluated **left to right**.

The operator **category precedence** is the following:

1. **Arithmetic** operators (highest precedence => evaluated first)
2. **Text string** operators
3. **Comparison** operators
4. **Pattern** operators
5. **Logical** operators (lowest precedence => evaluated last)

**Within each operator category** above, the **individual operator precedence** is the following:

#### **1. Arithmetic operators:**

1. “**^**” (exponential) (highest precedence => evaluated first)
2. “**-**” (change sign)
3. “**/**” and “**\***”

4. “\”
5. **Mod** (modulo)
6. “+” and “-” (minus) (lowest precedence => evaluated last)

## 2. Text string operators

1. “&” and “+”

## 3. Comparison operators:

1. = (highest precedence => evaluated first)
2. <>
3. <
4. >
5. <=
6. >= (lowest precedence => evaluated last)

## 4. Pattern operators:

1. “LIKE”

## 5. Logical operators:

1. “NOT” (highest precedence => evaluated first)
2. “AND”
3. “OR”
4. “XOR”
5. “Eqv”
6. “Imp” (lowest precedence => evaluated last)

My advice is that, unless you are **very sure** of the evaluation rules for an expression, you **should write parentheses** to indicate explicitly the **evaluation order** that you want for it.

## G.8 How do I use an SQL operation in an expression?

This chapter also answers the question:

- What is a Subquery?

SQL operations (“Subqueries”) can be used in the following scopes:

- **SQL expressions:**  
You **may** use a **Select** operation (click *F.7*), enclosed between parentheses, as a value.
- **VBA expressions:**  
This only applies to assigning the result of a SQL operation to an VBA variable (click *G.8.5*). To do it you can use special VBA built-in functions that allow to run SQL code. In this case, the rules for SQL code apply (click *F.6*).
- **Table “Design View” expressions:**  
You **cannot** use an **SQL** operation in field/record validation rule expressions, default field value expressions nor calculated field expressions. However, you can use an SQL operation in the “**Lookup**” property expressions of a Table/Form (click *D.11.4*).

You can use a **Select** operation as a **value** within an SQL expression as long as the **Select** produces **one** single **record** with **one** single **field**, and the **data type** of the **field** **matches the data type** required in the **expression** (plus a few other conditions). For example, if you write a **Select** operation that produces one single **Integer** value (i.e., one single record, with one single **Integer** field), it can be used in an **expression** in your SQL code as if it were an **Integer** value.

A **Subquery** is therefore a **Select operation** enclosed between parentheses that can be used as a **value** in an SQL **expression**. A Subquery **cannot** be a **Union** operation (although the **Select** operation may have an internal **Union** operation), nor a **Join** operation, nor a **Transform** operation.

When the **Subquery** is a **Select-total\_aggreg** over only one Table or Query, a good alternative may be to use a **domain aggregate function**: if you want to know more about them, you may click “[G.6.2 How do I use domain aggregate functions in an expression?](#)”.

If you want more detail about **Subqueries**, you may click:

- “[G.8.1 How do I use a Subquery in an SQL expression?](#)”
- “[G.8.2 What are the “EXISTS”, “ANY” and “ALL” Subquery operators?](#)”
- “[G.8.3 What is a correlated Subquery?](#)”
- “[G.8.4 What is an uncorrelated Subquery?](#)”

### **G.8.1 How do I use a Subquery in an SQL expression?**

Let me show you a simple example<sup>67</sup> of a **Subquery** over the Tables “T\_Capital\_Rainfall\_Q” from *D.6.5* and “T\_Capital\_Temps” from *F.10.5*:

```
SELECT Capital, District, Cal_year, Quart, Temp_Max
FROM T_Capital_Temps
WHERE Capital = (SELECT First(Capital) FROM T_Capital_Rainfall_Q)
```

You may see that the Subquery is highlighted in bold font, with **maroon** keywords and enclosed between **maroon** parentheses. This Subquery returns only **one record**, with only **one field** of data type **String**. This is a clear case in which the returned record-list is **equivalent** to a **variable** of data type **String**, which is what the expression is requiring.

In addition to requiring data type matching, using Subqueries has some other characteristics. Let me summarize the most relevant characteristics of Subqueries:

- A Subquery **can only be** a **Select operation**. A Subquery **cannot be** a **Union** operation, nor a **Transform** operation nor a **Join** operation.
- The Subquery **Select** operation can contain **an interior Union** operation, **Join** operation or another **Select** operation.
- The Subquery **Select** operation **must be enclosed in parentheses**.
- The data type of the **Select** operation result **must match** the data type expected by the expression or by the Subquery operator (click *G.8.2*).
- If the data type required by the expression is just **one value** (**Boolean**, **Double**,

<sup>67</sup> This is the Query “**G\_Subquery**” from file “**Company\_Database.accdb**”.

*String*, ...), the **Select** operation **must return only one record, with only one field, with an equivalent or compatible field type.**

- If the data type required by the expression is a **list** of values (e.g., to be used with the “**IN**” operator), the **Select** operation **must return a record-list with only one field, having an equivalent or compatible field type.**
- If the data type required by the expression is a **list** of values (e.g., to be used with the “**IN**” value operator), the **Select** operation **cannot return one record with several fields.**
- The **Select** operation of a Subquery **can** have parameters, **with or without** a “**PARAMETERS**” clause.
- According to the Microsoft documentation, you may nest up to 31 Subqueries. On my view, going beyond three makes readability quite poor.

**Uncorrelated Subqueries** (click “[G.8.4 What is an uncorrelated Subquery?](#)”) are extremely useful to restrict the record-list that you extract from your Tables, to be further processed in your Queries.

If you want to know more about this, you may click “[K.7.3.2 Why should I consider using uncorrelated Subqueries and/or domain aggregate functions in “WHERE” expressions over Tables?](#)”.

## **G.8.2 What are the “EXISTS”, “ANY” and “ALL” Subquery operators?**

You may click:

- “[G.8.2.1 What is the “EXISTS” Subquery operator?](#)”
- “[G.8.2.2 What is the “ANY” Subquery operator?](#)”
- “[G.8.2.3 What is the “ALL” Subquery operator?](#)”

### **G.8.2.1 What is the “EXISTS” Subquery operator?**

The “**EXISTS**” Subquery operator takes a **Subquery** as its operand and returns **True** if the **output** record-list of the Subquery has **one or more records**. It returns **False** if the **output** record-list has **no records**.

The way to write (syntax) the “**EXISTS**” operator is:

```
EXISTS Subquery
```

Let me show you a simple example<sup>68</sup> of the “**EXISTS**” Subquery operator over the Tables “T\_Capital\_Rainfall\_Q” from *D.6.5* and “T\_Capital\_Temps” from *F.10.5*:

```
SELECT Capital, Cal Year, Quart, Quart_Rainfall
FROM T_Capital_Rainfall_Q AS CRF
WHERE EXISTS (SELECT Capital
               FROM T_Capital_Temps
               WHERE Capital = CRF.Capital) ;
```

You may see that the Subquery is highlighted with **maroon** keywords and enclosed

---

<sup>68</sup> This is the Query “**G\_Subquery\_EXISTS**” from file “**Company\_Database.accdb**”.

between **maroon** parentheses. This Query returns the records from “T\_Capital\_Rainfall\_Q” whose value of the field “Capital” **exists** in the “Capital” field value-list in the Table “T\_Capital\_Temps”.

### G.8.2.2 What is the “ANY” Subquery operator?

The “**ANY**” Subquery operator **must** be **jointly** used with a Comparison operator. The “**ANY**” operator and the Comparison operator **jointly** perform a comparison between **one value** and **each of the values** returned by the **Subquery**. The **joint** result of “**ANY**” plus the Comparison operator is **True** when the Comparison operator returns **True** for **one of the values** returned by the **Subquery**. The **joint** result is **False** if the Comparison operator returns **False** for **all the values** returned by the **Subquery**.

The way to write (syntax) the “**ANY**” Subquery operator is:

```
expression() Comparison_Operator ANY Subquery
```

Let me show you a simple example<sup>69</sup> of the “**ANY**” Subquery operator over the Tables “T\_Capital\_Rainfall\_Q” from *D.6.5* and “T\_Capital\_Temps” from *F.10.5*:

```
SELECT Capital, Cal_Year, Quart, Quart_Rainfall
FROM T_Capital_Rainfall_Q
WHERE Quart_Rainfall > ANY (SELECT Temp_Max FROM T_Capital_Temps) ;
```

You may see that the Subquery is highlighted in bold font, with **maroon** keywords and enclosed between **maroon** parentheses. This Query returns the records from “T\_Capital\_Rainfall\_Q” whose value of the field “Quart\_Rainfall” is **larger** than **at least one** value in the “Quart\_Rainfall” field value-list in the Table “T\_Capital\_Temps”.

### G.8.2.3 What is the “ALL” Subquery operator?

The “**ALL**” Subquery operator **must** be **jointly** used with a Comparison operator. The “**ALL**” operator and the Comparison operator **jointly** perform a comparison between **one value** and **each of the values** returned by the **Subquery**. The **joint** result of “**ALL**” plus the comparison operator is **False** when the Comparison operator returns **False** for **one of the values** returned by the **Subquery**. The **joint** result is **True** if the Comparison operator returns **True** for **all the values** returned by the **Subquery**.

The way to write (syntax) the “**ALL**” Subquery operator is:

```
expression() Comparison_Operator ALL Subquery
```

Let me show you a simple example<sup>70</sup> of the “**ALL**” Subquery operator over the Tables “T\_Capital\_Rainfall\_Q” from *D.6.5* and “T\_Capital\_Temps” from *F.10.5*:

```
SELECT Capital, Cal_Year, Quart, Quart_Rainfall
FROM T_Capital_Rainfall_Q
WHERE Quart_Rainfall > ALL (SELECT Temp_Max FROM T_Capital_Temps) ;
```

You may see that the Subquery is highlighted in bold font, with **maroon** keywords and enclosed between **maroon** parentheses. This Query returns the records from “T\_Capital\_Rainfall\_Q” whose value of the field “Quart\_Rainfall” is **larger**

<sup>69</sup> This is the Query “G\_Subquery\_ANY” from file “Company\_Database.accdb”.

<sup>70</sup> This is the Query “G\_Subqueries\_ALL” from file “Company\_Database.accdb”.

than **all** the values in the “**Quart\_Rainfall\_Q**” field value-list in the Table “**T\_Capital\_Temps**”.

### G.8.3 What is a correlated Subquery?

A **correlated** Subquery is the one that uses a **value from** the **enclosing Select** operation in the **Subquery Select** operation.

Let me show you a simple example<sup>71</sup> of a **correlated** Subquery over the Tables “**T\_Capital\_Rainfall\_Q**” from *D.6.5* and “**T\_Capital\_Temps**” from *F.10.5*:

```
SELECT Capital, Cal_Year, Quart, Quart_Rainfall
FROM T_Capital_Rainfall_Q AS CRF
WHERE EXISTS (SELECT Capital
              FROM T_Capital_Temps
              WHERE Capital = CRF.Capital) ;
```

You may see that the Subquery is highlighted in bold font, with **maroon** keywords and enclosed between **maroon** parentheses. In this Query, the **value “CRF.Capital”** from the enclosing **Select** operation is being used in the **Select** operation of the Subquery. When the “**WHERE**” *Boolean* expression of the enclosing **Select** operation is being evaluated **for each of its records**, the **variable “CRF.Capital”** is replaced by the **value** it has in the record from the enclosing **Select** operation that is being evaluated. This implies that the Subquery has to be evaluated **as many times** as the number of **input records** of its enclosing **Select** operation. Correlated Subqueries are therefore considerably slower than uncorrelated Subqueries.

### G.8.4 What is an uncorrelated Subquery?

An **uncorrelated** Subquery is the one that **does not** use **any value from** the **enclosing Select** operation in the **Subquery Select** operation.

Let me show you a simple example<sup>72</sup> of a **uncorrelated** Subquery over the Tables “**T\_Capital\_Rainfall\_Q**” from *D.6.5* and “**T\_Capital\_Temps**” from *F.10.5*:

```
SELECT Capital, Cal_Year, Quart, Quart_Rainfall
FROM T_Capital_Rainfall_Q
WHERE Quart_Rainfall >= (SELECT Max(Temp_Min) FROM T_Capital_Temps)
```

You may see the Subquery is highlighted in bold font, with **maroon** keywords and enclosed between **maroon** parentheses. This Query produces the records from “**T\_Capital\_Rainfall\_Q**” that have a value of “**Quart\_Rainfall**” greater than the maximum value of “**Temp\_Min**” from the Table “**T\_Capital\_Temps**”.

Notice that contrary to what happens with correlated Subqueries, this uncorrelated Subquery is evaluated **only once**, and its result is used many times. Correlated Subqueries are therefore considerably slower than uncorrelated Subqueries.

### G.8.5 How do I use an SQL operation in a VBA variable assignment?

You **cannot** use an SQL operation as a variable in a VBA expression. However, you **can** assign the **result** of an SQL operation **to a VBA variable**, and then use the variable in any VBA expression. This provides basically the same functionality as using an SQL operation as a variable in a VBA expression. If you want to know more about this, you

<sup>71</sup> This is the Query “**G\_Subquery\_corr**” from file “**Company\_Database.accdb**”.

<sup>72</sup> This is the Query “**G\_Subquery\_uncorr**” from file “**Company\_Database.accdb**”.

may click “[K.9.3 How do I write a VBA function that reads database records?](#)”.

Also, you can use a **domain aggregate function** in a VBA expression. If you want more to know more about **domain aggregate functions**, you may click “[G.6.2 How do I use domain aggregate functions in an expression?](#)”.

## **G.9 How are numeric-like values internally represented and processed?**

### **G.9.1 How are numeric-like values internally represented?**

Values from **integer-like** data/field types (click [G.2.4](#)) are internally represented by the computer as **binary integers**. Values from **currency** data/field types (click [G.2.4](#)) are internally represented by the computer as **binary fixed-point** numbers. Values from **fractional-like** data/field types (click [G.2.4](#)) other than **currency** are internally represented by the computer as **binary floating-point** numbers.

If you want to know more about internal representation of data/field type values, such as number of bytes, range of values or precision, you may check the tables in:

- “[G.2.1 What VBA data types vs. Table field types-sizes are equivalent?](#)”
- “[G.2.2 What VBA data types vs. Table field types-sizes are not equivalent?](#)”

If you want an individual description of Table field types-sizes, you may click:

- “[D.4 How do I configure a Table field data type and size?](#)”.

In summary, values from **numeric-like** data/field types (click [G.2.4](#)) are internally represented by the computer as **binary integers**, **binary floating-point** numbers or **binary fixed-point** values. This implies that when you type-in a value it is required to do **decimal to binary** conversion, and when the computer displays a value it has to do **binary to decimal** conversion. If you want to know more about decimal/binary conversion, you may check the next section.

### **G.9.2 What is decimal/binary conversion?**

It is the process done by the computer to convert **from/to** the decimal numbers that you use **to/from** the binary numbers that the computer uses internally.

When you write a number, the usual form is using decimal numbering:

**...DCBA.abc...**

where each letter represents a **decimal digit** (i.e., an integer number from 0 to 9). The value it represents is:

$$\dots \mathbf{D} * 10^3 + \mathbf{C} * 10^2 + \mathbf{B} * 10^1 + \mathbf{A} * 10^0 + \mathbf{a} * 10^{-1} + \mathbf{b} * 10^{-2} + \mathbf{c} * 10^{-3} \dots$$

The above is a **decimal** or **base 10** number, because as you may see, the value is represented as the addition of **powers of 10**, each multiplied by a **decimal coefficient**.

It is also possible to use the same approach to represent a number with different values of the **base**. In particular, computers represent **internally all** the information they process as bits, this is, either a 1 or a 0. They also represent **numbers** (in most formats) as powers of two, this is in “**base 2**” or “**binary**” format. A number represented as



powers of two takes the form:

...DCBA.abc...

where each letter represents a **binary digit** (i.e., either a 0 or a 1). The value it represents is:

$$\dots D*2^3 + C*2^2 + B*2^1 + A*2^0 + a*2^{-1} + b*2^{-2} + c*2^{-3} \dots$$

The above is a **binary** or **base 2** number, because as you may see, the value is represented as the addition of **powers of 2**, each multiplied by a **binary coefficient**.

A relevant advantage of binary representation is faster number computations because the computer can process operations in hardware, using the Arithmetic and Logic Unit (“ALU”) of the processor.

### G.9.3 What are decimal/binary conversion rounding errors?

They are **rounding errors** caused by decimal/binary conversion.

Only the floating-point data/field types *Single*, *Double*, **Number-Single** and **Number-Double** cause rounding errors of decimal/binary conversion. The other data/field types do not have this problem.

The following table shows some examples of the **results** you will get when you try to store some large integer numbers into **single** data/field types:

G_Decimal_Binary_1	
expression	Result
CSng(2147483583)	2147483520
CSng(2147483584)	2147483648
CSng(2147483776)	2147483648
CSng(2147483777)	2147483904

as you may see, the **actual** stored value is **different** from the value you **wrote!**

You may check the above results writing the following Query<sup>73</sup>:

```

SELECT "CSng(2147483583)" AS expression, CSng(2147483583) AS Result
FROM T_Numbers WHERE Num=1
UNION
SELECT "CSng(2147483584)" AS expression, CSng(2147483584) AS Result
FROM T_Numbers WHERE Num=1
UNION
SELECT "CSng(2147483776)" AS expression, CSng(2147483776) AS Result
FROM T_Numbers WHERE Num=1
UNION
SELECT "CSng(2147483777)" AS expression, CSng(2147483777) AS Result
FROM T_Numbers WHERE Num=1

```

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click K.2.2).

Another example of decimal/binary conversion rounding errors is when you want to represent decimal **fractional** numbers in binary. The following table shows some examples of the **rounded** results you will get when you store **fractional** numbers in

<sup>73</sup> This is the Query “G\_Decimal\_Binary\_1” from file “Company\_Database.accdb”.

**single** data/field types:

G_Decimal_Binary_2	
expression	Result
CSng(.1)	0,100000001490116
CSng(.01)	9,99999977648258E-03
CSng(.001)	1,00000004749745E-03

You may check the above results writing the following Query<sup>74</sup>:

```

SELECT "CSng(.1)" AS expression, CSng(.1) AS Result
FROM T_Numbers WHERE Num=1
UNION
SELECT "CSng(.01)" AS expression, CSng(.01) AS Result
FROM T_Numbers WHERE Num=1
UNION
SELECT "CSng(.001)" AS expression, CSng(.001) AS Result
FROM T_Numbers WHERE Num=1

```

The **Double** and **Number-Double** data/field types also have rounding errors, but their precision is much higher than the one of **Single** and **Number-Single**. Therefore, it is easier to notice the rounding errors with **single** data/field types than with **double** ones.

Another interesting example. If I ask you what is the result of the following Query<sup>75</sup>:

```

SELECT Iif(5/10 - 4/10 - 1/10 = 0, "CORRECT", "WRONG!") AS Result_1
, Iif(.5 - .4 - .1 = 0, "CORRECT", "WRONG!") AS Result_2
FROM T_Numbers WHERE Num=1

```

you will most likely answer that it will be **two** “CORRECT”, if you are **not** aware of how decimal/binary rounding errors work, or **two** “WRONG!”, if you **are** aware of how decimal/binary rounding errors work. However, the result of the Query is “WRONG!” in field “Result\_1” and “CORRECT” in field “Result\_2”. The result “WRONG!” happens precisely because of decimal/binary rounding errors. The result “CORRECT” happens because the expression only has **constants**, and therefore it is correctly evaluated by the optimizer without losing precision.

If you want to know the cause of decimal/binary conversion rounding errors, you may check the next section.

#### **G.9.4 What is the cause of decimal/binary conversion rounding errors?**

The cause of decimal/binary conversion rounding errors is that the **floating-point** values you can represent in **decimal** notation and the **floating-point** values you can represent in **binary** notation **are not exactly the same**.

For integer values, if you want to store a large integer number in a single precision floating point data type, you **cannot** represent some values when you reach the maximum precision supported by **single**.

For fractional values, a fractional value that has an exact (finite) decimal representation may have a repeating (infinite) binary representation and vice versa. For example, the fraction 1/10 is an exact value in decimal (being “.1”), but in binary it is the **repeating**

<sup>74</sup> This is the Query “G\_Decimal\_Binary\_2” from file “Company\_Database.accdb”.

<sup>75</sup> This is the Query “G\_Decimal\_Binary\_3” from file “Company\_Database.accdb”.

number:

0.000110011...0011... (where the binary digits “0011” repeat infinite times)

Since the number of bits to store each value in a computer is obviously **finite**, the computer is forced to **round** the repeating representation of the value to the number of bits available in the data type that you are using to store it. This implies in practice that the computer is **changing** the value, and it is introducing a **conversion rounding** error. If you store “.1” decimal as a **double**, the conversion rounding error is in the order of  $-2.8E-17$ , which is very small. In spite of being very small, this error is enough so that some calculations (e.g., comparison operators) will not produce the result that you expect.

Using the **double** format used in MS-Access and VBA (under standard IEEE 754), the value “.1” decimal would be encoded as:

- **Sign** (1 bit): 0 (positive)
- **Exponent** (11 bits): 01111111 011  
(this is the value “-4” decimal encoded as 1019 in binary because of the exponent bias of 1023)
- **Mantissa** (phantom bit plus 52 actual bits. Only actual bits are shown):  
.10011001 10011001 10011001 10011001 10011001 10011001 1001  
(notice the leftmost “1” is missing, because it is the phantom bit)

## PART H. CUSTOMIZING THE APPEARANCE OF A QUERY/TABLE/FORM IN “DATASHEET VIEW”

Customizing the **appearance** of a Table/Query/Form in “**Datasheet View**” does not affect **stored data**, nor **data results** from Queries, in any way.

Customizing the **appearance** only changes the way **data is displayed** in a Table/Query/Form in “**Datasheet View**”.

You may click:

- “*H.1 How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?”*”
- “*H.2 How do I change row height, hide rows or change row order in a Table/Query/Form?”*”
- “*H.3 How do I change the order of columns that I see in a Table/Query/Form?”*”
- “*H.4 How do I hide/unhide columns in a Table/Query/Form?”*”
- “*H.5 How do I change the column headings in a Table/Query/Form?”*”
- “*H.6 How do I configure the formatting of column values in a Table/Query/Form?”*”
- “*H.7 How do I configure the column text alignment in a Table/Query/Form?”*”
- “*H.8 How do I show aggregate values (e.g., totals) in a Table/Query/Form?”*”
- “*H.9 How do I configure colors, fonts and other features of a Table/Query/Form?”*”

### H.1 How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?



You may click:

- “*H.1.1 How do I change the column width in a Table/Query/Form?”*”
- “*H.1.2 How do I freeze/unfreeze the columns in a Table/Query/Form?”*”

#### H.1.1 How do I change the column width in a Table/Query/Form?

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).


You can then **change** the **width** of **column(s)** in either of the following ways:

- Place the mouse over the **column name row**, right on the **division line** between two column name cells, and the mouse pointer will change to the horizontal drag “” icon. While the horizontal drag “” icon is shown, you **click-and-drag**, and you can make the column to the **left** of the icon narrower or wider.

You can change the width of several contiguous columns in one shot by first **selecting** several contiguous columns (click *B.5.2*) and then doing the same as indicated in the previous paragraph in one of the selected columns. The column width of **all** the selected columns will be changed to the width of the one you have dragged.

- Place the mouse on the **column name cell of the column** you want to change width,

right-click on it, and click on “**Field Width**” from the pop-up menu. This will open a box showing the current column width in “point” units. You can type-in the value you want or tick the checkbox “**Standard Width**”. When you are done, click on “**OK**” and the width of the column will be set to the value that you typed-in.

You can change the width of several contiguous columns in one shot by first **selecting** several contiguous columns (click B.5.2) and then **right-clicking** on any place of the selected columns. **Avoid** doing the right-click when the mouse pointer has changed to a **white cross** “

After right-clicking, select “**Field Width**” from the pop-up menu. This will open a box showing the current column width in “point” units. You can type-in the value you want or tick the checkbox “**Standard Width**”. When you are done, click on “**OK**” and the width of **all** the columns that you had selected will be set to the value that you typed-in.

In case the **formatted** representation of a value **does not fit** in the cell (because the formatted representation is larger than the cell’s width and height), then MS-Access will **not show** the value and will instead show the cell **filled** with “#” characters (e.g., “#####”). This is a nice feature, to avoid you “reading” a **wrong** field value just because it is partially hidden. This **does not apply** to **Short Text** nor *String* values: if a **Short Text** or *String* value is larger than the cell size, and cannot be shown as a whole, MS-Access will show **the part** of the text string **that fits** in the cell size. The reason is that **Short Text** or *String* values do not usually fit in the cell size, but still is very useful to **see them partially**.

If you wanted to see the columns in a different width for just a while, you can **discard all** your changes by **not saving** the Table/Query layout. When you close the Table/Query without having saved it, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you click on “**No**”. If you want to keep the current column width **permanently**, either previously **save** the Table/Query layout (click B.4.1.6) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you will have to **manually revert them** before closing the Form.

Changing the width of columns does not affect stored data and has no side effects.

### **H.1.2 How do I freeze/unfreeze the columns in a Table/Query/Form?**

Freezing columns in “**Datasheet View**” consists of showing them permanently by not making them subject to the horizontal scroll of columns. This is similar to freezing panels in Excel, in case you are familiar with it. All the frozen columns **must be contiguously** placed in the **leftmost** side of the Table.

Freezing columns **is extremely useful** when you cannot see all the columns in your MS-Access window, because it allows to show the one(s) that identify each record. I explain


this in the following two subsections:

- “H.1.2.1 How do I freeze the columns in a Table/Query/Form?”
- “H.1.2.2 How do I unfreeze all the columns in a Table/Query/Form?”

### H.1.2.1 How do I freeze the columns in a Table/Query/Form?

Open the Table/Query/Form in “**Datasheet View**” (click B.4.1.3).

Place the mouse on the **column name cell** of the column you want to freeze, right-click on it, and click on “**Freeze Fields**” from the pop-up menu. The column will be **moved** to the rightmost side of the Table/Query/Form and will be **frozen**.

You can freeze **several contiguous columns** in one shot by first **selecting** several contiguous columns (click B.5.2) and then **right-clicking** on any place of the selected columns. **Avoid** right-clicking when the mouse pointer has changed to a **white cross** “

After right-clicking, select “**Freeze Fields**” from the pop-up menu and of **all** the columns you had selected will be **moved** to the rightmost side of the Table and will be **frozen**.

You will normally want to keep the frozen columns **permanently**, so you have to **save** the Table/Query (click B.4.1.6). If you forgot to save the Table/Query and you close it, MS-Access will **ask**:

*“Do you want to save changes to the layout of table 'Table\_Name'?”*

and you should click on “**Yes**”. If you just wanted to freeze the columns for a while, **do not save** the Table/Query and click on “**No**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you will have to **manually revert them** before closing the Form.

Freezing columns does not affect stored data, has no side effects and **is considered a very good practice**.

### H.1.2.2 How do I unfreeze all the columns in a Table/Query/Form?

Open the Table/Query/Form in “**Datasheet View**” (click B.4.1.3).

Right-click over **any** column name cell and click on “**Unfreeze All Fields**” from the pop-up menu. This will unfreeze all fields. However, it will **not move them back** to the position they had before you froze them (in case you froze fields that were not placed on the leftmost side of the Table/Query/Form).

If you wanted to unfreeze the columns just for a while, you can **discard all** your changes by **not saving** the Table/Query layout. When you close the Table/Query without having saved its layout, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you should click on “**No**”. If you want to keep the columns **permanently unfrozen**, either previously **save** the Table/Query layout (click B.4.1.6) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just

wanted the changes for a while, you will have to **manually revert them** before closing the Form.

Unfreezing columns does not affect stored data and has no side effects.

## **H.2 How do I change row height, hide rows or change row order in a Table/Query/Form?**



You may click:

- “H.2.1 How do I change the height of all the rows in a Table/Query/Form?”
- “H.2.2 How do I hide rows in a Table/Query/Form?”
- “H.2.3 How do I change the sorting of rows in a Table/Query/Form?”

### **H.2.1 How do I change the height of all the rows in a Table/Query/Form?**

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

You can change the height of **all** the rows in a Table/Query/Form in either of the following ways:

- Place the mouse over the gray column to the left of the first field column, right on the division line between two rows, and the mouse pointer will change to the vertical drag “” icon. While the vertical drag “” icon is shown, you **click-and-drag**, and you can make all the rows in the Table/Query/Form narrower or wider.
- Place the mouse on the gray column just to the left of the first field column, right-click on it, and click on “**Row Height**” from the pop-up menu. This will open a box showing the **current** row height in “point” units. You can type-in the value you want or tick the checkbox “**Standard Height**”. When you are done click on “**OK**” and the height of all the rows will be changed to the value you typed-in.

If you wanted to see the rows in a different height for just a while, you can **discard all** your changes by **not saving** the Table/Query layout. When you close the Table/Query without having saved the layout, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you should click on “**No**”. If you want to keep the current row height **permanently**, either previously **save** the Table/Query (click *B.4.1.6*) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you will have to **manually revert them** before closing the Form.




In case the **formatted** representation of a value **does not fit** in the cell (because the formatted representation is larger than the cell’s width and height), then MS-Access will **not** show the value and will instead show the cell **filled** with “#” characters (e.g., “#####”). This **does not apply** to **Short Text** values: if a **Short Text** value is larger than the cell size, and cannot be shown as a whole, MS-Access will show the part of the text string that fits in the cells size.









Changing the height of rows does not affect stored data and has no side effects.

### H.2.2 How do I hide rows in a Table/Query/Form?

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

Then, configure filters to hide the records you want. To configure a filter, click on either of the icons “”, “ Selection ▾” or “ Advanced ▾” from the “**Sort & Filter**” Ribbon group in the “**Home**” Ribbon. You can configure simple filters, just based on a matching or not\_matching value, and also filters based on complex expressions.

You can alternate between applying and not applying the filter in either of the following ways:

- Clicking on either the “**Filtered**”/“**No Filter**” toggle icons in the “Navigation Bar”, which is located at the bottom of the Table/Query/Form pane. When the filter is being applied, the **Filtered** “ Filtered” icon is shown, and when the filter is **not** being applied the **Unfiltered** “ Unfiltered” icon is shown. Clicking on either icon will toggle between applying/not-applying the filter. If there is no filter set, the icon shown is “ No Filter”.
- Clicking on the **Toggle Filter** icons from the “**Sort & Filter**” Ribbon group from the **Home** Ribbon. When the filter is being applied, the “ Toggle Filter” icon is shown, and when the filter is **not** being applied the “ Toggle Filter” icon is shown. Clicking on either icon will toggle between applying/not-applying the filter. If there is no filter set, the icon shown is “ Toggle Filter”.

If you wanted to use the filters just for a while, you can **discard all** your changes by **not saving** the Table/Query layout. When you close the Table/Query without having saved its layout, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you should click on “**No**”. If you want to keep the current filter **permanently**, either previously **save** the Table/Query layout (click *B.4.1.6*) or click “**Yes**” upon the question above. Notice that when you open the Table/Query in “**Datasheet View**” it **always** starts with the filter **not** being applied.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you have to **manually revert them** before closing the Form.

Hiding/unhiding rows (i.e., applying filters) in “**Datasheet View**” does not affect stored data and has no side effects.

### H.2.3 How do I change the sorting of rows in a Table/Query/Form?





Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).


MS-Access allows you to sort the rows on ascending (or descending) order of the values of one, or more columns.

To **sort** the rows on the values of **one column**, you can do it in either of the following ways:

- Right-click anywhere in the column and then click on the sorting type you want from



the pop-up menu.

- Select the column by clicking on its header, and then click on a sorting icon (ascending order “ Ascending” or descending order “ Descending”) from the “**Sort & Filter**” Ribbon group in the “**Home**” Ribbon.
- Click on the sorting “” icon placed on the right side of the column header (e.g., the icon on the right side of “”) and then click on the sorting type that you want from the pop-up menu. Notice that this way of sorting will **not** work on crosstab Queries (Queries with a **Transform** operation).

Regardless of the way you configure sorting, you can revert it by click on the Remove Sorting “ Remove Sort” icon from the “**Sort & Filter**” Ribbon group in “**Home**” Ribbon.

**Ascending** sorting means that rows are sorted, **top to bottom, a/A to z/Z** in **text** fields, **smallest** (negative numbers) **to** largest in **integer** and **fractional** fields, **True/Yes/On** or **ticked** to **False/No/Off** or **unticked** in **Yes/No** or **Boolean** fields, and **oldest to most recent** in **Date/Time** or **Date** fields. **Descending** sorting means the opposite. If you want to know more detail about sorting criteria, you may click “*F.7.12.1 How are the different data/field types ordered by the “ORDER BY” clause?*”.

To **sort** the rows based on the values of **several columns**, you should sort the rows in **reverse** column order. This is, if you want the rows to be sorted **first** on the values of column **A**, **then** on the values of **B**, and **finally** on the values of **C**, you **first** sort on the values of **C**, **then** on **B** and **finally** on **A**: the resulting sorting is what you wanted.

Regardless of the way you sort the rows, **every** column **heading** you have used to sort the rows will **show a vertical arrow** on its rightmost side indicating if that column has been configured as ascending order (upward arrow “”) or descending order (downward arrow “”).

If you wanted to see the rows with this sorting just for a while, you can **discard all** your changes by **not saving** the Table/Query layout. When you close the Table/Query without having saved the layout, MS-Access will **ask**:

“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”

and you should click on “**No**”. If you want to keep the current row sorting **permanently**, either previously **save** the Table/Query layout (click *B.4.1.6*) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you have to **manually revert them** before closing the Form.

Changing the sorting of **Query** rows in “**Datasheet View**” and then saving the Query **causes a problem** in MS-Access, in case the Query has **parameters**. The problem is that the Query parameters will be requested **twice** (or more times), creating confusion and potential for errors. My advice is you **never save the row sorting** you have done **in a Query in “Datasheet View”**. If you want to change permanently the sorting or rows, do it with an “**ORDER BY**” clause in the SQL code of the Query (click *F.7.12*). If you want to know more about the problem of requesting a parameter twice, you may click “*J.11.8 How do I fix a Query requesting the same parameter twice (or more times)?*”.

Changing the sorting of rows in “**Datasheet View**” does not affect stored data and has no side effects (except for the specific problem explained above).

### **H.3 How do I change the order of columns that I see in a Table/Query/Form?**

It is different to change the **order of columns** that you see in “**Datasheet View**” than changing the **order of the actual database elements** that support each column. Let us see the three specific cases:

- **Tables**

It is different to change the **order of Table columns** that you see in “**Datasheet View**” than changing the **order of Table fields** in “**Design View**”. If you want to know more about these differences, you may click “*H.3.2 How is it related a Table’s column order and its field order in “Design View”?*”.

- **Queries**

It is different to change the **order of Query columns** that you see in “**Datasheet View**” than changing the **order of Query output fields** in “**Design View**” or in “**SQL View**”. If you want to know more about these differences, you may click “*H.3.3 How is it related a Query’s column order and its field order in “Design View” and “SQL View”?*”.

- **Forms**

It is different to change the **order of Form columns** that you see in “**Datasheet View**” than changing the **order of Table fields** of the origin Table of the Form. If you want to know more about these differences, you may click “*H.3.4 How is it related a Forms’ column order and shown/hidden columns and the ones of its associated Table?*”.

If you want to know more about this, you may click:

- “*H.3.1 How do I change the order of columns in a Table/Query/Form?*”
- “*H.3.2 How is it related a Table’s column order and its field order in “Design View”?*”
- “*H.3.3 How is it related a Query’s column order and its field order in “Design View” and “SQL View”?*”
- “*H.3.4 How is it related a Forms’ column order and shown/hidden columns and the ones of its associated Table?*”

#### **H.3.1 How do I change the order of columns in a Table/Query/Form?**

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

**Select** the column that you want to move (change order) by clicking on its field name (the column heading). The column you wanted should now be shaded, and this tells you that it has been selected. You can now drag-and-drop the column **heading cell** to change the column position left or right to the place you want.

You can move **several** contiguous columns in one shot by selecting several columns (click *B.5.2*) instead of only one.

If you wanted to see the columns with this order just for a while, you can **discard all** your changes by **not saving** the Table/Query design. When you close the Table/Query without having saved its design, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you should click on **“No”**. If you want to keep the current column order **permanently**, either previously **save** the Table/Query (click *B.4.1.6*) or click **“Yes”** upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you have to **manually revert them** before closing the Form.

Notice that if you want to change permanently the column order in a Table, I advise you actually change its field order in **“Design View”** because the column order in **“Datasheet View”** will be **lost** the next time that you do a design change. Likewise, if you want to change permanently the column order in a Query, I advise you change the output field order in **“SQL View”** because the column order in **“Datasheet View”** will be lost the next time that you do a design change or SQL change.

Changing the order of columns in **“Datasheet View”** does not affect stored data and has no side effects.

### **H.3.2 How is it related a Table’s column order and its field order in “Design View”?**

When you create a Table, its column order in **“Datasheet View”** is the same as its field order in **“Design View”**.

If you change the Table column order in **“Datasheet View”**, this **will not affect** the Table field order in **“Design View”**, even if you save the Table and make the changes permanent.

However, **each time** you change the field order of a Table in **“Design View”**, once you save the Table, the field order you have configured will be **also** applied to the column order in **“Datasheet View”**.

There is nothing specifically wrong in having a different order of the Table’s columns in **“Datasheet View”** than the field order in the Table’s records. However, if you forget about this difference, it may lead to making errors when coding your SQL Queries.

### **H.3.3 How is it related a Query’s column order and its field order in “Design View” and “SQL View”?**

When you create a Query, its column order in **“Datasheet View”**, its field order in **“Design View”**, its field order in **“SQL View”** and its field order in the **“Access SQL Editor”** are **all the same**.

The field order in **“Design View”** and **“SQL View”** are always **the same**. Therefore, if you change the field order in either of them, the same field order will appear in the other one.

If you change the **column** order in **“Datasheet View”** this **will not** change the **field** order in **“Design View”**, **“SQL View”** nor in the **“Access SQL Editor”**.

If you change the **field** order in “**Design View**” or “**SQL View**” this will **also** change the **column** order in “**Datasheet View**”, but it **will not** change the **field** order in the “**Access SQL Editor**”. Notice however that doing this will cause that the Query’s SQL code in MS-Access and in the “**Access SQL Editor**” will be **different** (click *L.8.4*). I therefore advise you **never** change the field order using “**Design View**” or “**SQL View**”.

If you change the **field** order in the “**Access SQL Editor**”, this will also change the **column** order in “**Datasheet View**” and the **field** order in “**Design View**” and “**SQL View**”.

There is nothing specifically wrong in having a different order of the Query’s columns in “**Datasheet View**” than the field order in the Query’s records. However, if you forget about this difference, it may lead to making errors when coding your SQL Queries.

### **H.3.4 How is it related a Forms’ column order and shown/hidden columns and the ones of its associated Table?**

The order of columns and the columns shown/hidden in a Form in “**Datasheet View**” is totally unrelated to the order of columns and columns shown in its associated Table (in “**Datasheet View**”), and totally unrelated to the order of fields in its associated Table (in “**Design View**”). The only requirement is that the Table **field** associated to a Form column has **not been removed** from the Table.

You can change the order of columns and hide/unhide columns in a Table or Form, and it will have no effect in its associated Form or Table (respectively).

## **H.4 How do I hide/unhide columns in a Table/Query/Form?**

**Hiding/unhiding** some **columns** that you see in a Table/Query/Form in “**Datasheet View**” is **very different** from **suppressing/adding fields** in the Table/Query **records**. Hiding some columns is just a matter of changing the **appearance** of the Table/Query/Form in “**Datasheet View**” and it **does not** have any side effect on other Tables, Forms, Queries nor in any other element of your database. However, **suppressing/adding fields** in the **records** of the Table/Query **will have side effects**.

In the following four sections I explain how to hide/unhide Table/Query/Form **columns** in “**Datasheet View**”:

- “*H.4.1 How do I hide the columns that I see in a Table/Query/Form?”*”
- “*H.4.2 How do I unhide columns in a Table/Query/Form?”*”
- “*H.4.3 How are related a Table/Query’s shown/hidden columns and its fields in other views?”*”
- “*H.4.4 How is it related a Forms’ column order and shown/hidden columns and the ones of its associated Table?”*”

If you rather want to **add/delete fields** to/from a **Table**, you may click:

- “*B.6.1.5 How do I add Table fields in “Design View”?*”
- “*B.6.1.8 How do I delete Table fields in “Design View”?*”

If you rather want to **suppress/add fields** in a **Query's output records**, you may click:


- “F.7.5 What are the output fields (“SELECT” clause) of a Select?”
- “F.8.4 What are the output fields of a Join?”
- “F.9.3 What are the output fields of a Union?”
- “F.10.3 What are the output fields of a Transform?”

#### **H.4.1 How do I hide the columns that I see in a Table/Query/Form?**

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

You can then hide column(s) in either of the following ways:

- Place the mouse on the **column name cell** of the column you want to hide, right-click on it, and click on “**Hide Fields**” from the pop-up menu. The column will be hidden.

You can hide several contiguous columns in one shot by first **selecting** several contiguous columns (click *B.5.2*) and then **right-clicking** on any place of the selected columns. **Avoid** right-clicking when the mouse pointer has changed to a **white cross** “”, because if you do it, the column range will be de-selected, the current cell will be selected, and a different pop-up menu will appear.

After having right-clicked, select “**Hide Fields**” from the pop-up menu and **all** the columns you had selected will be hidden.

- Right-click over any column name cell and click on “**Unhide fields**” from the pop-up menu. This will open a dialogue-box where all the column names are shown, each with a checkbox to its left. You can now **tick** or **untick** each of the column names. If you **untick** a column name, that column will be hidden. If you **tick** it the column will be unhidden. When you are done **ticking/unticking** column names, click on the “**Close**” button. As you may see, you can also use this method to **unhide** columns.

If you wanted to hide the columns just for a while, you can **discard all** your changes by **not saving** the Table/Query design. When you close the Table/Query without having saved its design, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you should click on “**No**”. If you want to keep the current hidden columns **permanently**, either previously **save** the Table/Query design (click *B.4.1.6*) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you will have to **manually revert them** before closing the Form.

Hiding columns in “**Datasheet View**” does not affect stored data and has no side effects, but it may be difficult for you to create new Table/Form records with hidden columns.

#### **H.4.2 How do I unhide columns in a Table/Query/Form?**

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

Right-click over any column name cell and click on “**Unhide Fields**” from the pop-up menu. This will open a box where all the column names are shown, each with a checkbox to its left. You can now **tick** or **untick** each of the column names. If you **untick** a column name, that column will be hidden. If you **tick** it the column will be unhidden. When you are done **ticking/unticking** column names, click on the “**Close**” button. As you may see, you can also use this method to **hide** columns.

If you wanted to unhide the columns just for a while, you can **discard all** your changes by **not saving** the Table/Query design. When you close the Table/Query without having saved its design, MS-Access will **ask**:

*“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”*

and you should click on “**No**”. If you want to keep the current unhidden columns **permanently**, either previously **save** the Table/Query layout (click *B.4.1.6*) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard changes: if you just wanted the changes for a while, you will have to **manually revert them** before closing the Form.

Unhiding columns in “**Datasheet View**” does not affect stored data and has no side effects.

#### **H.4.3 How are related a Table/Query’s shown/hidden columns and its fields in other views?**

The shown/hidden columns in a **Table** in “**Datasheet View**” are totally unrelated to its fields in “**Design View**”.

If you change the Table’s shown/hidden columns in “**Datasheet View**”, this **will not affect** the Table’s fields in “**Design View**”, even if you save the Table **layout** and make the changes permanent.

The columns shown/hidden in a Query result in “**Datasheet View**” are totally unrelated to its fields in “**Design View**” and “**SQL View**”.

If you change the Query’s shown/hidden columns in “**Datasheet View**”, this **will not affect** the Query’s fields in “**Design View**” nor in “**SQL View**”, even if you save the Query **layout** and make the changes permanent.

#### **H.4.4 How is it related a Forms’ column order and shown/hidden columns and the ones of its associated Table?**

The order of columns and the columns shown/hidden in a Form in “**Datasheet View**” is totally unrelated to the order of columns and columns shown in its associated Table (in “**Datasheet View**”), and totally unrelated to the order of fields in its associated Table (in “**Design View**”). The only requirement is that the Table **field** associated to a Form column has **not been removed** from the Table.

You can change the order of columns and hide/unhide columns in a Table or Form, and it will have no effect in its associated Form or Table.



## H.5 How do I change the column headings in a Table/Query/Form?

Changing the column headings that you see in a Table/Query/Form in “**Datasheet View**” is **very different** from **changing the field names** in the Table/Query records. Changing the column heading that you see is just a matter of changing the **appearance** of the Table/Query/Form in “**Datasheet View**” and it **does not** have any side effect on other Tables, Forms, Queries nor in any other element of your database. However, **changing the field names** in the records of the Table/Query **will have side effects**.

If you want to change the heading of Table/Query/Form **columns** in “**Datasheet View**”, you may click:

- “H.5.1 How do I change a Table’s column headings?”
- “H.5.2 How do I change a Query’s column headings?”
- “H.5.3 How do I change a Form’s column headings?”
- “H.5.4 How are related a Form’s column headings and the ones of its associated Table?”

If you rather want to **change the field names** in a **Table**, you may click:

- “B.6.1.7 How do I reconfigure my Table fields in “Design View”?”

If you rather want to change the **change the field names** in a **Query’s output records**, you may click:

- “F.7.5 What are the output fields (“SELECT” clause) of a Select?”
- “F.8.4 What are the output fields of a Join?”
- “F.9.3 What are the output fields of a Union?”
- “F.10.3.2 What are the output field names of a Transform?”

### H.5.1 How do I change a Table’s column headings?

Remind that changing the Table’s **column headings** that you see in “**Datasheet View**” is **very different** from changing the Table’s **field names** (click B.6.1.7).

If the Table is not opened in “**Design View**”, you either **open** it (click B.4.1.3) in “**Design View**” or **change** its **view-type** (click B.4.1.4) to “**Design View**”.

Select the field associated to the column heading you want to change by clicking on the corresponding field row.

Click on the row named “**Caption**” in the field properties, placed at the bottom sub-pane, in the “**General**” tab.

In the box to the right of “**Caption**”, type-in the label you want to be shown in this column in “**Datasheet View**”.

You may repeat the process for as many fields/columns that you want.

Once you are done with your Table configuration, **save** (click B.4.1.6) your Table **design**. You may then **close** the Table (click B.4.1.7) or change it to “**Datasheet View**”


(click *B.4.1.4*).

Changing the Table column headings in “**Datasheet View**” does not affect stored data and has no side effects.

### **H.5.2 How do I change a Query’s column headings?**

Remind that changing the Query’s **column headings** that you see in “**Datasheet View**” is **very different** from changing the Query’s **field names** (click *H.5*).

If the Query is not opened in “**Design View**”, you either **open** it (click *B.4.1.3*) in “**Design View**” or **change** its **view-type** (click *B.4.1.4*) to “**Design View**”.

Then, you can either click on the **Property Sheet**  icon from the “**Query Tools / Design**” contextual Ribbon, or else, right-click anywhere on the “Query pane”, and click on “**Properties**” from the pop-up menu. This will show the “**Property Sheet**”.

Click on the field name corresponding to the Query column heading that you want to change. The field names are in the first row of the “Query design grid”, placed in the bottom sub-pane. In case the field name that you want is not currently shown, use the horizontal scrollbar at the bottom of the bottom sub-pane.

Click on the row “**Caption**”, within the “**Property Sheet**” placed at the right of the “Query pane”, in the “**General**” tab. In the box to the right of “**Caption**”, type-in the column heading you want to be shown in the column associated to this field in “**Datasheet View**”.

You may repeat the process for as many column headings that you want to change.


Once you are done with your Query configuration, **save** (click *B.4.1.6*) your Query **design**. You may then **close** the Query (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).


Changing the Query result column headings in “**Datasheet View**” does not affect stored data and has no side effects.

### **H.5.3 How do I change a Form’s column headings?**

Remind that changing the **Form’s column headings** that you see in “**Datasheet View**” is **very different** from changing the supporting **Table’s field names** (click *H.5*).

Open the Form in “**Datasheet View**” (click *B.4.1.3*).


Then, you can either click on the **Property Sheet**  icon from the “**Form Tools / Datasheet**” contextual Ribbon, or else, right-click anywhere on the “Form pane”, and click on “**Properties**” or “**Form Properties**” from the pop-up menu. This will show the “**Property Sheet**” on the right side of the “Form pane”<sup>76</sup>.

Click on the drop-down  icon in the element box, at the top of the “**Property Sheet**”, and click on the label element corresponding to the field name you want to change. The label element is the one with the field name followed by the suffix “**\_label**”. Click on the tab “**Format**”, and then on the row “**Caption**”. In the box to the right of “**Caption**”, type-in the column heading you want to be shown in the column associated to this field

---

<sup>76</sup> You can also show the “Property Sheet” of a Form from its “Design View”.

in “**Datasheet View**”.

Remind that you can toggle between property default order and alphabetical order by clicking on the **A-Z** “” icon placed at the top-right corner of the “**Property Sheet**”.

You may repeat the process for as many column headings that you want to change.

Once you are done with your Form configuration, **save** (click *B.4.1.6*) your Form **layout**. You may then **close** the Form (click *B.4.1.7*).

Changing the Form column headings in “**Datasheet View**” does not affect stored data and has no side effects.

#### **H.5.4 How are related a Form’s column headings and the ones of its associated Table?**

The column headings in a Form in “**Datasheet View**” will be the same as the ones in its associated Table, if you created the Form from the Table using the “**Form Wizard**” from the “**Create**” Ribbon.

However, if you **later** change the **column heading(s)** of the Form (click *H.5.3*) or of its associated Table (click *H.5.1*), the column labels in the Form are totally unrelated to column heading in its associated Table. The only requirement is that the Table **field** associated to a Form column has **not been removed** from the Table.

#### **H.6 How do I configure the formatting of column values in a Table/Query/Form?**

You do it by assigning specific values to the **formatting properties** of the column in “**Datasheet View**”. To locate the formatting properties of a Table click *H.6.3.1*, of a Query result click *H.6.3.2* and of a Form click *H.6.3.3*.

The most important formatting property of a Table/Query/Form **column** is “**Format**”. You locate the “**Format**” property (click *H.6.3*) corresponding to the **column** you want to configure. You then **type-in** the **predefined** or **custom** format you want, or else, you **choose** one of the **predefined** format options using the drop-down menu. It is very important to notice that you can type-in **any applicable predefined format**, even if it is **not shown in the drop-down menu**.

Changing the display formatting of a Table/Query/Form column in “**Datasheet View**” **does not affect stored data** and has no side effects. **However**, in case the **formatted** representation of a value **does not fit** in the cell (because the formatted representation is larger than the cell’s width and height), then MS-Access will **not** show the value and will instead show the cell **filled** with “#” characters (e.g., “#####”). This **does not apply** to **Short Text** or **String** values: if a **Short Text** or **String** value is larger than the cell size, and cannot be shown as a whole, MS-Access will show the part of the text string that fits in the cells size.

If you want to configure **predefined** or **custom** formats, and/or **find** the formatting **properties**, you may click:

- “*H.6.1 How do I configure predefined column formatting in a Table/Query/Form?”*”
- “*H.6.2 How do I configure custom column formatting in a Table/Query/Form?”*”

- “*H.6.3 How do I find the column formatting properties in a Table/Query/Form?”*”

### **H.6.1 How do I configure predefined column formatting in a Table/Query/Form?**

The most important formatting property of a Table/Query/Form **column** in “**Datasheet View**” is “**Format**”. To locate the formatting properties of a Table click *H.6.3.1*, of a Query result click *H.6.3.2* and of a Form click *H.6.3.3*. You first locate the “**Format**” property corresponding to the **column** you want to configure. You then **type-in** the **predefined** or **custom** format that you want, or else, you **choose** one of the predefined format options shown in the drop-down menu. It is very important to notice that you can type-in **any predefined format**, even if it is **not shown in the drop-down menu**.

The **predefined** formats that are shown in the drop-down menu depend on the **field type** of the **column**. I will explain the predefined formats for the following four cases:

- **Short Text** field type  
Click “*H.6.1.1 What “Format” options are available in Short Text and Long Text fields?*”.
- **Yes/No** field type  
Click “*H.6.1.2 What “Format” options are available in Yes/No fields?What “Format” options are available in Yes/No fields?*”.
- **Number, Large Number, Currency, and Date/Time** field types  
Click “*H.6.1.3 What “Format” options are available in Number, Large Number, Currency and Date/Time fields?*”.
- **Calculated** field type  
Click “*H.6.1.4 What “Format” options are available in Calculated fields?*”.

#### **H.6.1.1 What “Format” options are available in Short Text and Long Text fields?**

If you write “**Standard**” as the value of the “**Format**” “**Datasheet View**” property of a **Short Text** or **Long Text** field, the values shown in “**Datasheet View**” in this column will be right-aligned.

**No other predefined** formats are available (only custom formatting is available).

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

#### **H.6.1.2 What “Format” options are available in Yes/No fields?**

The formatting of the **Yes/No** field type in **Table “Datasheet View”** is **different from** the other field types, because it depends on the configuration of its “**Display Control**” property in the “**Lookup**” tab of the **Table** field.

Having clarified this, the predefined formatting options for a **Yes/No field** are the

following:

- **Table field** with “**Display Control**” set to “**Check Box**”  
The **Yes/No** value is **always** shown as a **checkbox**, **no matter** what you configure in its “**Format**” property.
- **Query/Form field directly** coming from a **Table** field that has “**Display Control**” set to “**Check Box**”  
The value is **always** shown as a **checkbox**, **no matter** what you configure in its “**Format**” property.  
If you want to change the formatting of such a **Query** field, the solution is as simple as adding in the Query code “**OR True**” to the Table field name in the “**SELECT**” **expression**, so the Query field is now the result of a **Boolean** operator. In this way, the Query field now corresponds to the case of the next bullet point.
- **Table field** with “**Display Control**” set to “**Text Box**” or “**Combo Box**”, or **Query/Form field directly** coming from a **Table** field that has “**Display Control**” set to “**Text Box**” or “**Combo Box**”, or **Query field** resulting from **any Boolean** operator/function  
The options for the “**Format**” property are:
  - Blank: values will be shown as “**-1**” or “**0**”.
  - “**True/False**”: values will be shown as “**True**” or “**False**”.
  - “**Yes/No**”: values will be shown as “**Yes**” or “**No**”.
  - “**On/Off**”: values will be shown as “**On**” or “**Off**”.
 Notice that **you can** configure the above options for the “**Format**” property, even if they are **not shown** in its drop-down menu: you just have to **type-in** the corresponding text.

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

### H.6.1.3 What “**Format**” options are available in Number, Large Number, Currency and Date/Time fields?

For the purpose of formatting in “**Datasheet View**”, the values of these five field types are **all equally interpreted** as a floating-point number. For **Date/Time**, the integer part is the date-part (0 being 30-dec-1989) and the fractional part is the time-part.

The field types **Number** and **Currency** have the additional formatting property “**Decimal Places**” that the field type **Date/Time** does not have. The additional property “**Decimal Places**” is located right below the “**Format**” property.

The predefined formats available for these four field types are:

- **Default** format (when the “**Format**” property is blank)  
In this case, the applied formatting depends on the field type, as follows:
  - **Number** field type: value is shown with “**General Number**” format.
  - **Currency** field type: value is shown with “**Currency**” format.

- **Date/Time** field type: value is shown with “**General Date**” format.
- “**General Number**” predefined format  
Numbers written in normal decimal notation with all its decimal digits. Negative numbers are prefixed by “-”.
- “**Currency**” predefined format  
Notice that there may be some confusion because the **name** of the “**Currency**” predefined format **is the same** as the **Currency** field type. You should remind that the **Currency** field type and the “**Currency**” predefined format are **very** different things, although (unfortunately) they have the same name.  
The “**Currency**” format is similar to “**General number**”, but adding thousands separation characters, showing the number of decimal digits indicated in the “**Decimal Places**” property and appending a blank followed by the currency character (e.g., \$, €, ...) defined in your system. If the value of “**Decimal Places**” is “**Auto**”, then two decimal digits are shown.
- “**Fixed**” predefined format  
Similar to “**General number**” but showing the number of decimal digits indicated in the “**Decimal**” property. If the value of “**Decimal Places**” is “**Auto**”, then two decimal digits are shown.
- “**Standard**” predefined format  
Similar to “**General number**” but adding thousands separation characters and showing the number of decimal digits indicated in “**Decimal Places**”. If the value of “**Decimal Places**” is “**Auto**”, then two decimal digits are shown.
- “**Percent**” predefined format  
Similar to “**General number**”, but the value shown is the stored value multiplied by 100, decimal digits shown are the value of the “**Decimal Places**” property and the “%” character is appended. If “**Decimal Places**” is set to “**Auto**”, then two decimal digits are shown.
- “**Scientific**” predefined format  
Number is shown in decimal exponential notation (i.e., **2.0376E+02**). Decimal digits shown are the value of the “**Decimal Places**” property. If “**Decimal Places**” is set to “**Auto**”, then two decimal digits are shown.
- “**General Date**” predefined format  
Values with time-part equal **0:00:00** are shown as “**m/d/yyyy**” where “**d**” is day number, “**m**” is month number and “**yyyy**” is year number. The time-part is not shown. Example:  
**1/15/2019**  
Values with date-part equal to 30-December-1899 (i.e., integer part equal “0”) are shown as “**h:nn:ss xx**” where “**h**” is hour (**0** to **12**), “**nn**” is minutes, “**ss**” is seconds and “**xx**” is either **AM** or **PM**. Day, month and hour less than 10 are shown with one digit. Minutes and seconds are always two digits. Example: **1:09:07 PM**  
Values with date-part and time-part different from the two values above are shown as:  
**m/d/yyyy h:nn:ss xx**

where the letters mean the same as in the previous two cases. Example:

**1/15/2019 1:09:07 PM**

- “**Long Date**” predefined format  
Only the date-part is shown as “**Weekday, Month d, yyyy**” where Weekday and Month are shown as names and day “**d**” and year “**yyyy**” are shown as numbers. Day numbers less than 10 are shown with **one** digit. Time-part is **not** shown. Example, “**Tuesday, January 1, 2019**”.
- “**Medium Date**” predefined format  
Only the date-part is shown as “**dd-mmm-yy**” where “**mmm**” are the first three letters of each month’s name. Day is always shown with **two** digits. Year is shown with two digits if the first two are “**20**” and with four digits otherwise. Example, “**15-Jan-19**”.
- “**Short Date**” predefined format  
Only the date-part is shown as “**m/d/yyyy**” where “**m**” is month number, “**d**” is day number, and “**yyyy**” is year number. Day and month less than 10 are shown with one digit. Year is always shown with four digits. Example: “**1/15/2019**”.
- “**Long Time**” predefined format  
Only the time-part is shown as “**h:nn:ss xx**” where “**h**” is hour (**0** to **12**), “**nn**” is minutes, “**ss**” is seconds and “**xx**” is either **AM** or **PM**. Hour less than 10 is shown with one digit. Minutes and seconds are always two digits. Example: “**1:09:07 PM**”.
- “**Medium Time**” predefined format  
Only the time-part is shown as “**h:nn xx**” where “**h**” is hour (**0** to **12**), “**nn**” is minutes and “**xx**” is either **AM** or **PM**. Hour less than 10 is shown with one digit. Minutes are always two digits. Example: “**1:09 PM**”.
- “**Short Time**” predefined format  
Only the time-part is shown as “**h:nn**” where “**h**” is hour (**0** to **23**) and “**nn**” is minutes. Hour less than 10 is shown with one digit. Minutes are always two digits. Example: “**13:09**”.

Notice that some of the predefined formats above are **not displayed** in the pop-up menu shown when clicking in the rightmost side of the “Format” row, but still can use them if you type them in.

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

#### **H.6.1.4 What “Format” options are available in Calculated fields?**

The predefined formats available in “**Datasheet View**” for a **Calculated** field are the ones corresponding to the **equivalent field type** of its “**Result Type**” property, for all the field types. If you want to know the predefined formats for all the **actual** field types,



you may click:

- “H.6.1.1 What “Format” options are available in Short Text and Long Text fields?”
- “H.6.1.2 What “Format” options are available in Yes/No fields?”
- “H.6.1.3 What “Format” options are available in Number, Large Number, Currency and Date/Time fields?”

## **H.6.2 How do I configure custom column formatting in a Table/Query/Form?**

You have to be careful with custom formatting “**Datasheet View**” because if you use strange formats you may have problems when copying data from MS-Access and pasting as text in other programs. I advise you use custom formatting as you want, as long as you **do not add additional text**. Adding additional text will almost for sure create problems when pasting as text in other programs.

The way to configure custom formatting is by writing in the “**Format**” property a text string composed of characters that convey formatting instructions to MS-Access. You locate the “**Format**” property corresponding to the **column** you want to configure. To locate the formatting properties of a Table click *H.6.3.1*, of a Query result click *H.6.3.2* and of a Form click *H.6.3.3*.

The custom formatting most frequently used is for **dates**, and you will see in the corresponding table below that MS-Access allows **very flexible** custom formatting for dates.

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

If you want to know each of the formatting characters, and what is their formatting effect, you may click:

- “H.6.2.1 How do I configure custom formatting for all field types?”
- “H.6.2.2 How do I configure custom formatting for Short Text fields?”
- “H.6.2.3 How do I configure custom formatting for Number, Large Number and Currency fields?”
- “H.6.2.4 How do I configure custom formatting for Date/Time field type?”

### **H.6.2.1 How do I configure custom formatting for all field types?**

This subsection presents the **common** formatting **characters** for **all** field types. In addition to these, there are **specific** formatting **characters** for **some** field types, that I present in the next two subsections:

- “H.6.2.2 How do I configure custom formatting for Short Text fields?”
- “H.6.2.3 How do I configure custom formatting for Number, Large Number and Currency fields?”

The following table shows the formatting characters to write “**Datasheet View**” custom

formatting strings that are common to **all field types**:

Character	Description (for all field types)
*	When used, the character immediately after the asterisk (*) becomes a fill character (a character used to fill blank spaces). Access normally displays text as left-aligned and fills any area to the right of the value with blank spaces. You can add fill characters anywhere in a format string. When you do so, MS-Access fills any blank spaces with the specified character.
Blank space, + - \$ € ()	Used to insert blank spaces, math characters “+”, “-”, financial symbols “\$”, “€”, and parentheses as needed anywhere in your format strings. If you want to use other common math symbols, such as slash (\ or /) and the asterisk (*), enclose them between double quotes. Note that you can place these characters anywhere in the format string.
"Literal text"	Use double quotes to enclose any text that you want to be displayed to users.
\	Display the character that immediately follows. Typically used when you want to display a character that has some special meaning like “*”. This is the same as enclosing a character between double quotes.
[color]	Displays in color all values in a section of your format. You must enclose the color name in brackets and use one of these names: <b>black</b> , <b>blue</b> , <b>cyan</b> , <b>green</b> , <b>magenta</b> , <b>red</b> , <b>yellow</b> , or <b>white</b> .

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

### H.6.2.2 How do I configure custom formatting for Short Text fields?

The following table shows the formatting characters to write “**Datasheet View**” custom formatting strings for **Short Text** field type. Remind that you can also use the common formatting characters for all field types shown in “*H.6.2.1 How do I configure custom formatting for all field types?*”.

Character	Description (for Short Text field type)
@	Displays any available character for its position in the format string. If MS-Access places all characters in the underlying data, any remaining placeholders appear as blank spaces. For example, if the format string is @@@@ and the underlying text is <b>ABC</b> , the text is left-aligned with two leading blank spaces.
&	Displays any available character for its position in the format string. If MS-Access places all characters in the underlying data, any remaining placeholders display nothing. For example, if the format string is &&&& and the text is <b>ABC</b> , only the left-aligned text is displayed.
!	Displays text left justified. You must use this character at the start of any format string.

Character	Description (for Short Text field type)
<	Displays all text in lowercase. You must use this character at the beginning of a format string, but you can precede it with an exclamation point (!).
>	Displays all text in uppercase. You must use this character at the beginning of a format string, but you can precede it with an exclamation point (!).

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

### H.6.2.3 How do I configure custom formatting for Number, Large Number and Currency fields?

The following table shows the formatting characters to write “**Datasheet View**” custom formatting strings for **Number**, **Large Number** and **Currency** field types. Remind that you can also use the common formatting characters for all field types shown in “*H.6.2.1 How do I configure custom formatting for all field types?*”.

Character	Description (for Number, Large Number and Currency field types)
#	Displays a digit. Each instance of the character represents a position for one digit. If no value exists in a given position, MS-Access displays a blank space. It can also be used as a placeholder. For example, if you apply the format <b>#,###</b> and enter a value of <b>45</b> in the field, <b>45</b> is displayed. If you enter 12,145 in a field, MS-Access displays <b>12,145</b> , even though you defined only one placeholder to the left of the thousands separator.
0	Used to display a digit. Each instance of the character represents a position for one digit. If no value exists in a position, MS-Access displays a zero ( <b>0</b> ).
<b>Decimal separator</b> “.” (period)	Indicates where you want MS-Access to place the separator character between the whole and decimal parts of a <b>Number</b> or <b>Currency</b> field. For non-English versions, decimal separator may vary and is set in the regional settings in Windows.
<b>Thousands separator</b> “,” (comma)	Indicates where you want MS-Access to place the separator character between the thousands part of a <b>Number</b> or <b>Currency</b> field. For non-English versions, thousands separator may vary is set in the regional settings in Windows.

Character	Description (for Number, Large Number and Currency field types)
E+, E- or e+, e-	Used to display values in scientific (exponential) notation. Use this option when the predefined scientific format doesn't provide sufficient room for your values. Use <b>E+</b> or <b>e+</b> to display values as positive exponents, and <b>E-</b> or <b>e-</b> to display negative exponents. You must use these placeholders with other characters. For example, suppose that you apply the format <b>0.000E+00</b> to a <b>numeric-like</b> field and then enter 612345. MS-Access displays <b>6.123E+05</b> . MS-Access first rounds the number of decimal places down to three (the number of zeros to the right or left of the decimal separator). Next, MS-Access calculates the exponent value from the number of digits that fall to the right (or left, depending on your language settings) of the decimal separator in the original value. In this case, the original value would have put 612345 (five digits) to the right of the decimal point. For that reason, MS-Access displays <b>6.123E+05</b> , and the resulting value is the equivalent of $6.123 \times 10^5$ .
!	Used to force the left alignment of all values. When you force left alignment, you cannot use the <b>#</b> and <b>0</b> digit placeholders, but you can use placeholders for text characters.
%	Used as the last character in a format string. Multiplies the value by 100 and displays the result with a trailing percent sign.

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

#### H.6.2.4 How do I configure custom formatting for Date/Time field type?

The following table shows the formatting characters to write custom formatting strings for **Date/Time** field type. Remind that you can also use the common formatting characters for all field types shown in “*H.6.2.1 How do I configure custom formatting for all field types?*”.

Character	Description (for Date/Time field type)
<b>Date separator</b>	Separator character between days, months, and years. Use the separator defined in the Windows regional settings. For example, in English (U.S.), use a slash “/” or hyphen “-” character.
<b>Time separator</b>	Separator character between hours, minutes, and seconds. Use the separator defined in the Windows regional settings. For example, in English (U.S.), use a colon “:” character.
<b>c</b>	Displays the “ <b>General Date</b> ” predefined format (click <i>H.6.1.3</i> ).
<b>d</b> or <b>dd</b>	Displays the day of the month as one or two digits.
<b>ddd</b>	Displays the first three letters of the name of day of the week.
<b>dddd</b>	Displays the complete name of the day of the week.
<b>dddd</b>	Displays the “ <b>Short Date</b> ” predefined format (click <i>H.6.1.3</i> ).

Character	Description (for Date/Time field type)
<b>dddddd</b>	Displays the “ <b>Long Date</b> ” predefined format (click <i>H.6.1.3</i> ).
<b>w</b>	Displays a number that corresponds to the day of the week ( <b>1 to 7</b> ).
<b>ww</b>	Displays a number that corresponds to the week of the year ( <b>1 to 53</b> ).
<b>m or mm</b>	Displays the <b>month</b> as either a <b>one-digit</b> or <b>two-digit</b> number.
<b>mmm</b>	Displays the first <b>three letters</b> of the <b>name</b> of the <b>month</b> .
<b>mmmm</b>	Displays the <b>complete name</b> of the <b>month</b> .
<b>q</b>	Displays the <b>number</b> of the current calendar <b>quarter</b> ( <b>1 to 4</b> ). For example, if you hire a worker in May, MS-Access will display <b>2</b> as the quarter value.
<b>y</b>	Displays the <b>day</b> of the year, <b>1 to 366</b> .
<b>yy</b>	Displays the <b>last two digits</b> of the <b>year</b> .
<b>yyyy</b>	Displays the <b>year</b> as four digits in the range <b>0100 to 9999</b> .
<b>h or hh</b>	Displays the <b>hour</b> as <b>one</b> or <b>two</b> digits.
<b>n or nn</b>	Displays <b>minutes</b> as <b>one</b> or <b>two</b> digits.
<b>s or ss</b>	Displays <b>seconds</b> as <b>one</b> or <b>two</b> digits.
<b>tttt</b>	Displays the “ <b>Long Time</b> ” predefined format (click <i>H.6.1.3</i> ).
<b>AM/PM</b>	<b>Twelve-hour clock</b> with the <b>uppercase</b> letters “ <b>AM</b> ” or “ <b>PM</b> ”, as appropriate.
<b>am/pm</b>	<b>Twelve-hour clock</b> with the <b>lowercase</b> letters “ <b>am</b> ” or “ <b>pm</b> ”, as appropriate.
<b>A/P</b>	<b>Twelve-hour clock</b> with the <b>uppercase</b> letter “ <b>A</b> ” or “ <b>P</b> ”, as appropriate.
<b>a/p</b>	<b>Twelve-hour clock</b> with the <b>lowercase</b> letter “ <b>a</b> ” or “ <b>p</b> ”, as appropriate.
<b>AMPM</b>	<b>Twelve-hour clock</b> with the appropriate <b>morning/afternoon</b> designator as defined in the regional settings of Windows.

Notice that in non-English versions of MS-Access, some of the characters above may be **translated**. For example, in the Spanish version, the character “**y**” becomes “**a**” for “**año**”. Formatting characters may differ from the above ones in the MS-Access settings, but will stay the same in the VBA formatting functions, even in foreign language versions of MS-Access.

Once you are done with your Table/Query/Form configuration, **save** (click *B.4.1.6*) your Table/Query/Form design. You may then **close** the Table/Query/Form (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the column formatting in “**Datasheet View**” does not affect stored data and has no side effects.

### **H.6.3 How do I find the column formatting properties in a Table/Query/Form?**

The way to **find** the column formatting properties in “**Datasheet View**” is **specific** of a Table, Query result or Form, as I explain in the following subsections:

- “*H.6.3.1 How do I find a Table’s column formatting properties?*”

- “H.6.3.2 How do I find a Query’s column formatting properties?”
- “H.6.3.3 How do I find a Form’s column formatting properties?”

### H.6.3.1 How do I find a Table’s column formatting properties?

Open the Table in “**Design View**” (click B.4.1.3).

Select the field associated to the column you want to format by clicking on the corresponding field row.

Find the row named “**Format**”, in the field properties placed at the bottom sub-pane, in the “**General**” tab. In the cell to the right of “**Format**”, type-in the predefined/custom format that you want for this field. You can also right-click on the rightmost part of this box and click on a predefined format from the drop-down menu. However, notice that the drop-down menu **does not show all** of the predefined formats that you can use.

In case this is a **Number**, **Large Number** or **Currency** field, you will find the property “**Decimal Places**” right below “**Format**”. In the cell to the right of “**Decimal Places**”, either type-in the number of decimal digits you want for this field, or else, right-click on the rightmost part of this box and click on a value from the drop-down menu.

Find the row named “**Text Align**”, in the field properties placed at the bottom sub-pane, in the “**General**” tab. In the cell to the right of “**Text Align**”, either type-in the value you want for this field, or else, right-click on the rightmost part of the cell and click on a value from the drop-down menu.

You may repeat the process for as many Table fields that you want.


Once you are done with your Table configuration, **save** (click B.4.1.6) your Table design. You may then **close** the Table (click B.4.1.7) or change it to “**Datasheet View**” (click B.4.1.4).

Changing the value formatting of a Table’s column that is shown in “**Datasheet View**” does not affect stored data and has no side effects.

### H.6.3.2 How do I find a Query’s column formatting properties?

Open the Query in “**Design View**” (click B.4.1.3).

In case “**Design View**” is not shown, it is most likely because this is a **Union** Query: to **fix** this you may click “K.4.7 Why should I enclose the outermost Union operation in a Select operation?”.

Then, you can either click on the **Property Sheet**  icon from the “**Query Tools / Design**” contextual Ribbon, or else, right-click anywhere on the “Query pane”, and click on “**Properties**” from the pop-up menu. This will show the “**Property Sheet**” on the right side of the “Query pane”.

To configure the formatting properties of a given Query column, click on the corresponding **output field name** in the first row of the “Query design grid” placed in the bottom sub-pane of the “Query pane”. This will show the properties of that **output** field in the “**Property Sheet**”.

In the “**Property Sheet**”, find the row named “**Format**”. In the cell to the right of “**Format**”, type-in the predefined/custom format you want for this field. You can also

right-click on the rightmost part of this cell and click on a predefined format from the drop-down menu. However, notice that the drop-down menu **does not show all** the predefined formats that you can use.

In case this is a **Number, Large Number** or **Currency** field, you will find the property “**Decimal Places**” right below “**Format**”. In the cell to the right of “**Decimal Places**”, either type-in the number of decimal digits you want for this field, or else, right-click on the rightmost part of this box and click on a value from the drop-down menu.

Notice that Query columns **do not have** a “**Text Align**” property.

You may repeat the process for as many Query **output** fields that you want.


Once you are done with your Query configuration, **save** (click *B.4.1.6*) your Query design. You may then **close** the Query (click *B.4.1.7*) or change it to “**Datasheet View**” (click *B.4.1.4*).

Changing the value formatting of a Query’s column that is shown in “**Datasheet View**” does not affect stored data and has no side effects.

When writing your SQL Queries, **remind** to **qualify all the output field names** in your SQL Query code. Otherwise, all the formatting configuration of the Query columns that you have made for “**Datasheet View**” **will be lost** when you modify the Query SQL code. If you want to know more about this, you may click “*K.4.5 Why should I qualify all the outermost output field names of my Queries?*”.

### **H.6.3.3 How do I find a Form’s column formatting properties?**

Open the Form in “**Datasheet View**” (click *B.4.1.3*).

Then, you can either click on the **Property Sheet** “” icon from the “**Form Tools / Datasheet**” contextual Ribbon, or else, right-click anywhere on the “Form pane”, and click on “**Properties**” or “**Form Properties**” from the pop-up menu. This will show the “**Property Sheet**” on the right side of the “Form pane”<sup>77</sup>.

You can see that the “**Property Sheet**” shows at its top **an element box** with a drop-down menu to select the Form element for which you want to show its properties. Click on the name of the column whose formatting properties you want from the drop-down menu. Now select the “**Format**” tab in the “**Property Sheet**”.

In the “**Property Sheet**”, find the row named “**Format**”. In the cell to the right of “**Format**” type-in the custom/predefined format that you want for this column. You can also right-click on the rightmost part of this cell and click on a predefined format from the drop-down menu. However, notice that the drop-down menu **does not show all** the predefined formats that you can use.

In case that this is a **Number, Large Number** or **Currency** field, you will find the property “**Decimal Places**” right below “**Format**”. In the cell to the right of “**Decimal Places**”, either type-in the number of decimal digits you want for this field, or else, right-click on the rightmost part of this box and click on a value from the drop-down menu.


Find the row named “**Text Align**”. In the cell to the right of “**Text Align**”, either type-in the value you want for this column, or else, right-click on the rightmost part of this box

---

<sup>77</sup> You can also show the “Property Sheet” of a Form from its “Design View”.



and click on a value from the drop-down menu.

Remind that you can toggle between property default order and alphabetical order by clicking on the **A-Z** “” icon placed at the top right corner of the “**Property Sheet**”.

You may repeat the process for as many Form columns that you want.

Save the Form **layout** by right-clicking on the Form tab and clicking on “**Save**” from the pop-up menu. You can then **close** the Form or close the “**Property Sheet**” to introduce data into it.

Changing the value formatting of a Form’s column that is shown in “**Datasheet View**” does not affect stored data and has no side effects.




## **H.7 How do I configure the column text alignment in a Table/Query/Form?**

You may click:

- “*H.7.1 How do I configure a Table/Form’s column text alignment?*”
- “*H.7.2 How do I configure a Query’s column text alignment?*”

### **H.7.1 How do I configure a Table/Form’s column text alignment?**

For Tables and Forms you configure each “**Datasheet View**” column’s text alignment in either of the following ways:

- Select the column and click on the corresponding **text align** icon: left “”, center “” or right “” from the “**Home**” Ribbon.
- Configure a custom format in the “**Format**” property: click “*H.6.2 How do I configure custom column formatting in a Table/Query/Form?*”.
- Configure the “**Text Align**” property. You can find the “**Text Align**” property of each field/column in the same way as the “**Format**” property: click “*H.6 How do I configure the formatting of column values in a Table/Query/Form?*”).

You can configure one of the following options for the property “**Text Align**”:

- “**General**” (default option)  
Aligns the values of columns with **Short Text** field type to the **left** and the values of all the other field types to the **right**.
- “**Left**”  
Aligns each value to the **left** of the column.
- “**Center**”  
**Centers** each value in the column.
- “**Right**”  
Aligns each value to the **right** of the column.
- “**Distribute**”  
**Distributes** the characters of each value across all the column width, introducing the required spacing between the characters.

If you want the changes to be permanent, save the Table/Form by right-clicking on its tab and clicking on “**Save**” from the pop-up menu. You can then introduce data in the Table/Form pane or **close** the Table/Form pane.

Changing the Table/Form text alignment that is shown in “**Datasheet View**” does not affect stored data and has no side effects.

### **H.7.2 How do I configure a Query’s column text alignment?**

Query results columns in “**Datasheet View**” do not have the “**Text Align**” property, and the text align icons from the “**Home**” Ribbon will not work (they will be shaded). Therefore, the only way to configure the text alignment in a Query column is by configuring a custom formatting in the corresponding “**Format**” property.

For Query results my advice is that you **keep the default alignment**. This will make the **text strings left justified**, and **all other** field types **with a different justification** (centered, right, ...). This will allow you to **detect in just one glance** that a number, date, or other field types have been converted to *String*. This is a quite frequent mistake that can cause a number of errors. Detecting this in a glance is extremely useful.

The default alignment for **Short Text** fields is **left** alignment. If you want a **Short Text** field **right** aligned, you may you write “**Standard**” in its “**Format**” property.

The default alignment for **Number** fields is **right** alignment. If you want a **Number** field **left** aligned, you may write an exclamation mark “**!**” its “**Format**” property.

If you want to know more about configuring custom formatting, you may click “*H.6.2 How do I configure custom column formatting in a Table/Query/Form?*”.

### **H.8 How do I show aggregate values (e.g., totals) in a Table/Query/Form?**

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

You then click on the **Total** “ $\Sigma$ ” icon from the “**Home**” Ribbon. This will show an additional “aggregate” row at the end of the Table/Query/Form. This additional aggregate row will be labeled “**Total**” in the leftmost cell. If you click again on the Total “ $\Sigma$ ” icon this additional aggregate row will be hidden.

In case you use a **filter** (click *H.2.2*) to **only show some rows**, the value in the aggregate additional row will be computed **over the rows currently shown**. This is, the rows that have been **hidden** as a consequence of the filter, will be **completely ignored** for the result computed in the aggregate row. Combining filters and the aggregate values row is **very useful** to compute different aggregated results over the specific groups of records that you want.

You can select different types of aggregate values (total, average, ...) for each field within the aggregate row. To do this, click on the left side of **any** cell in the additional aggregate row (including the cell labeled “**Total**”, if you want) and you will see a drop-down menu where you can select the type of aggregate value you want. The **aggregate value options** shown in the drop-down menu depend on the field type of the column, as I explain in the following subsections:

- “*H.8.1.1 How do I show aggregate values in Short Text and Yes/No fields?*”

- “H.8.1.2 How do I show aggregate values in Date/Time field?”
- “H.8.1.3 How do I show aggregate values in Number, Large Number and Currency fields?”

If you wanted to see the **aggregate** row for just a while, you can **discard it** by **not saving** the Table/Query layout. When you close the Table/Query without having saved it, MS-Access will **ask**:

“Do you want to save changes to the layout of table/query 'Table/Query\_Name'?”

and you click on “**No**”. If you want to keep the **aggregate** row **permanently**, either previously **save** the Table/Query layout (click *B.4.1.6*) or click “**Yes**” upon the question above.

Notice that for the case of **Forms**, it is **not possible** to discard the “**Total**” row: if you just wanted it for a while, you have to **manually remove it** before closing the Form.

Adding the aggregate row to a Table/Query/Form in “**Datasheet View**” does not affect stored data and has no side effects.

You can also compute aggregate rows in your SQL Queries in a very flexible way (several aggregate rows, selective aggregate, show the row in a different position, ...). If you want to know more about this, you may click:

- “K.6.3 How do I produce totals in addition to individual results?”

#### **H.8.1.1 How do I show aggregate values in Short Text and Yes/No fields?**

Click on the left side of **the** cell that you want in the additional aggregate row (including the cell labeled “**Total**”) and click on the aggregate value option that you want from the pop-up menu.

The “**Datasheet View**” aggregate value options for **Short Text** and **Yes/No** fields are:

- **None:**  
This is the default value. If selected, will clear the aggregated value shown in the cell.
- **Count:**  
Will show in the cell the **count** of **non-Null** values **currently listed** in this column.

#### **H.8.1.2 How do I show aggregate values in Date/Time field?**

Click on the left side of **the** cell that you want in the additional aggregate row (including the cell labeled “**Total**”) and click on the aggregate value option that you want from the pop-up menu.

The “**Datasheet View**” aggregate value options for **Date/Time** fields are the ones available for **Short Text** and **Yes/No** fields, plus the following **additional** ones:

- **Average:**  
Will show in the cell the **average** of all the **non-Null** values **currently listed** in this column.
- **Maximum:**  
Will show in the cell the **maximum** value in the rows **currently listed** in this

column.

- **Minimum:**

Will show in the cell the **minimum** value in the rows **currently listed** in this column.

### **H.8.1.3 How do I show aggregate values in Number, Large Number and Currency fields?**

Click on the left side of **the** cell that you want in the additional aggregate row (including the cell labeled “**Total**”) and click on the aggregate value option that you want from the pop-up menu.

The “**Datasheet View**” aggregate value options for **Number**, **Large Number** and **Currency** fields are the ones available for **Date/Time** fields, plus the following **additional** ones:

- **Sum:**

Will show in the cell the **addition** of all the **non-Null** values **currently listed** in this column.

- **Standard Deviation:**

Will show in the cell the **standard deviation** of all the **non-Null** values **currently listed** in this column. Remind that you need at least two values to get a standard deviation result.

- **Variance:**

Will show in the cell the **variance** of all the **non-Null** values **currently listed** in this column. Remind that you need at least two values to get a variance result.

## **H.9 How do I configure colors, fonts and other features of a Table/Query/Form?**

Open the Table/Query/Form in “**Datasheet View**” (click *B.4.1.3*).

You then select the font type, font size, font style (bold, italic, ...) and/or the font color that you want by using the corresponding icons from the “**Text Formatting**” Ribbon group in the “**Home**” Ribbon.

You can also configure the borders, shadowing and other appearance of cells.

You can also configure the colors used for the alternate rows.

The selected changes are applied to the whole Table/Query/Form and cannot be selectively applied to individual columns or rows.

Remind that for the case of Tables and Forms, you can also configure the text alignment of each individual column (click *H.7.1*).

If you wanted to see this **formatting** for just a while, you can **discard it** by **not saving** the Table/Query/Form layout. When you close the Table/Query/Form without having saved it, MS-Access will **ask**:

“*Do you want to save changes to the layout of table/query 'Table/Query\_Name'?*”

and you click on “**No**”. If you want to keep this **formatting permanently**, either

previously **save** the Table/Query/Form layout (click *B.4.1.6*) or click “**Yes**” upon the question above.

Changing the Table/Query/Form appearance that is shown in “**Datasheet View**” does not affect stored data and has no side effects.

## PART I. EVOLVING MY DATABASE DESIGN

You may click:

- “[I.1 Why would I want to improve/modify my database design?](#)”
- “[I.2 Why should I be so careful with any change to the database design?](#)”
- “[I.3 How do I find all the dependent objects on a given database object?](#)”
- “[I.4 What are the side effects of modifying my Table fields?](#)”
- “[I.5 What are the side effects of modifying my Tables?](#)”
- “[I.6 What are the side effects of modifying my Queries?](#)”
- “[I.7 What are the side effects of modifying my user-defined VBA functions?](#)”
- “[I.8 What are the side effects of modifying my Relationships, Forms and/or Reports?](#)”

### I.1 Why would I want to improve/modify my database design?

Because you may want to fix errors, introduce more functionality, or adapt the database to changing environment and needs.

In theory, you carefully do a perfect initial database design when you create the database, and then you just use it as such forever. In practice, this **never** happens.

To start with, the most frequent way to do a database design is by doing it incrementally. You start creating some Tables and Relationships. You paste/input some test data to check if everything seems right. You then do some modifications to the Tables and Relationships, and you add a couple more Tables. You paste/input some more test data to check if everything seems right. You repeat this cycle a few times, until you consider that the database design is sufficiently sound. You then remove all data used for testing and input the correct initial data. The database is then operational, and you begin using it.

After the database is in operation, **you will also be doing, for sure**, some **modifications** to the database design. You may be for example adding more fields to the existing Tables, adding new Tables and Relationships, changing field names, field types, Relationships or **Key** fields. Modifications are sometimes motivated because you want either more functionality or adjust the current one. Other times, modifications are motivated because the environment (e.g., accounting regulation, privacy laws, your business expands, ...) changes.

### I.2 Why should I be so careful with any change to the database design?

Because **most** changes on your database design, **even seemingly minor**, will imply that **you will have to do adjustments** over other elements of the database (e.g., stored data, fields, Tables, Relationships, Queries and/or Forms). The reason for this is that database elements are **interdependent** among themselves. For example, if you change the name of a Table, all the Queries that use that Table will **crash**, because they will have the old Table name into their SQL code, and the Table with the original name does not exist

anymore.

You need to **understand well both** the **interrelations** between database elements (called “**dependencies**”) and the specific malfunctions (called “**side effects**”) that will happen on the dependent elements when you modify the element on which they depend. If you **understand well dependencies and side effects**, you will be able to **plan and execute the changes** in your database design **properly** and **without** introducing any **malfunctions** in the process.

My advice is that before making **any** modification on **any** database element (fields, Tables, Relationships, Queries or Forms), you check the corresponding chapter “**What are the sides effect of...**” clicking in **Part I**.

If you want to understand better the concepts of **dependencies** and **side effects**, you may click:

- “**I.2.1 What are database element dependencies?**”
- “**I.2.2 What are side effects?**”

### **I.2.1 What are database element dependencies?**

A **dependency** means that when you modify a database element this causes a **malfunction** on **other elements** that are said to **depend** on the element that has been modified.

The database **elements** in any database design are **very interrelated**. **Most** modifications you make on an element will cause **malfunctions** on its **dependent** elements. There is also a **cascading effect**: if you modify an element that causes its **K** dependent elements to malfunction, **each** of the **K** elements will in turn affect a number of elements that depend on it, and so on. With this cascading effect, the consequences of one modification may **propagate** causing malfunctions on **many** of your database elements. Therefore, when you modify an element, you have some elements that are **directly dependent** on it, but you may also have many more elements that are **indirectly dependent** on it.

For example, if you change the name of a given Query, all the Queries that make use of this Query will **crash**, because they cannot find the former Query name (direct dependence). Also, **all** other Queries that make use of **any** of these Queries that now **crash**, will also **crash** (indirect dependence). The same applies to further Queries making use of these ones (indirect dependence again), and so on.

**Adding** a new **Table, Form, Query, Report** or user-defined VBA **function** to the database design does **not** cause any **side effect**. You can therefore add as many as you need without experiencing any problem. **Adding** a new **Relationship** will restrict the values of the slave Tables, but this is not a side effect, and rather it is actually what you want. Therefore, **adding** a new **Relationship** does **not** cause any **side effect**.

However, **modifying** an **existing** database element has **side effects** on most cases. The following is a **list of elements** that have a **direct dependency** on each database element type:

- **Table** modifications **may** have direct side effects on:
  - All **Relationships** in which the Table is involved
  - All **Forms** based on the Table



- All **Reports** based on the Table
- All **Queries** that use the Table
- All **Tables** and **Forms** with a **drop-down menu** that has the Table name in its “**Row Source**” property
- All **user-defined VBA functions** and all **Forms’ VBA Event Subroutines**, that use the Table
- **Query** modifications **may** have direct side effects on:
  - All **Forms** based on the Query
  - All **Reports** based on the Query
  - All **Queries** that use the Query
  - All **Tables** and **Forms** with a **drop-down menu** that has the Query name in its “**Row Source**” property
  - All **user-defined VBA functions** and all **Forms’ VBA Event Subroutines**, that use the Query
- User-defined **VBA function** modifications **may** have direct side effects on:
  - All **Tables** that use the function in a **Calculated** field
  - All **Queries** that use the function
  - All **user-defined VBA functions** and all **Forms’ VBA Event Subroutines**, that use the function

Remind that these are only the **direct** dependencies, and that the cascading effect will propagate the malfunction to **many** other elements that have an **indirect** dependence on the modified element.

My advice is that before making **any** modification on **any** database element (fields, Tables, Relationships, Queries or Forms), you check the corresponding chapter “**What are the sides effect of...**” clicking in **Part I**.

### **I.2.2 What are side effects?**

**Side effects** are the specific **malfunctions** that a modification on an element may cause on its dependent elements. Some examples of side effects are:

- The dependent Query **crashes**.
- The dependent function **crashes**.
- The dependent Query requests a **non-existing** parameter.
- An **Insert operation** inserts **wrong** values in its target Table.
- A pop-up menu in the dependent Table is **not shown**.
- A pop-up menu in the dependent Table does not show **the values you expect**.

The specific side effect caused by modifying an element varies case by case, and it depends on the element type, and on the specific modification that has been done to the element.

My advice is that before making **any** modification on **any** database element (fields, Tables, Relationships, Queries or Forms), you should check the corresponding chapter “**What are the sides effect of...**” (click **Part I**).

### I.3 How do I find all the dependent objects on a given database object?

You do it using an **object dependency tool**.

MS-Access has a built-in **object dependency tool** that you can find in the “**Database Tools**” Ribbon. However, this tool just **does not do the job**. To start with, if you write your Queries in SQL, this built-in tool will not be able to find their dependencies. You could then think to use this tool only to track Table/Form dependencies. Even for this restricted usage there are problems: the tool function “**Objects on what I depend**” does not work properly and the tool will include master Tables as dependent objects, which on my view is misleading because it should only include slave Tables as dependent objects.

My advice is therefore not to use the built-in tool and rather install a third-party add-in **object dependency tool**. One such tool you may want to consider is:

<http://www.accessdependencychecker.com/>

### I.4 What are the side effects of modifying my Table fields?

You may click:

- “I.4.1 What are the side effects of adding a field to a Table?”
- “I.4.2 What are the side effects of deleting a Table field?”
- “I.4.3 What are the side effects of changing the name of a Table field?”
- “I.4.4 What are the side effects of modifying the properties of a Table field?”
- “I.4.5 What are the side effects of changing the order of Table fields?”

#### I.4.1 What are the side effects of adding a field to a Table?

**Adding** a new **field** to a **Table** (click *B.6.1.5*) causes side effects if you are using “**SELECT \***” over the Table, you have Forms over the Table or you have **Insert** operations into the Table. Side effects will affect the following database elements:

- **Queries with “SELECT \*” over the Table**

The result of the **Select** operation will additionally produce the field. This is what you may want in some cases. However, the additional field may propagate through other “**SELECT \***” higher up the SQL operation hierarchy and cause some of the problems listed in the following bullet points.

You **fix** this by **editing all such Queries** and replace the “**\***” by an explicit list of the corresponding Table fields wherever it is needed: see the problem description in the following bullet points.

- **Queries with “SELECT \*” over the Table in a Union operation**

The Queries with “**SELECT \***” will **crash** because the “**SELECT \***” will additionally produce the field, thus having a **different number** of fields than **the other input record-list** of the **Union** operator. Remind that both **input record-lists** to a **Union** operator **must** have the same number of fields.

You **fix** this by **editing all such Queries** and either:

- Replace the “\*” by an explicit list of the corresponding Table fields.
- Modify **all** the other **input** record-lists of the **Union** operations to match the additional field.

- **Queries with a “SELECT \*” over the Table, and the Select is used in an Insert-many-records operation**

The Queries will (most likely) **crash**. In the case of **using** the optional list of target Table field names, the **number** of fields will now **not** be the same. In case of **omitting** the optional list of target Table field names, the newly added field most likely does not exist in the target Table, and this **crashes** the said Queries. In case the newly added field exists in the target Table, this is most likely an unintended coincidence, and the **Insert operation** will insert wrong values in that same-name field: this error may be quite difficult to detect.

You **fix** this by **editing all such Queries** and either:

- Replace the “\*” by an explicit list with the corresponding Table fields.
- Modify the target Table to also include the newly added field (in case this is coherent with your database design).

If you want to know more about **Insert** operations, you may click “*F.13.2 What is an Insert operation and how do I write it?*”.

- **Queries with Insert-one-record operation without the optional list of target Table field names with the Table as its target Table**

The Query will **crash** because having added the field to the target Table causes that the number of “**VALUES**” expressions in the **Insert** operation is **different** from the number of fields in the target Table.

You **fix** this by **editing all such Queries** and correcting the corresponding **Insert** operations. If you want to know more about **Insert** operations, you may click “*F.13.2 What is an Insert operation and how do I write it?*”.

- **Queries with an Insert-many-records operation, with the Table as its target Table**

The Query will insert the default value (or **Null** if no default value) in the field in all the inserted records. If this is what you want, this is fine.

Otherwise, you **fix** this by **editing all such Queries** and correcting the corresponding **Insert** operations. If you want to know more about **Insert** operations, you may click “*F.13.2 What is an Insert operation and how do I write it?*”.

- **User-defined VBA functions and Forms’ VBA Subroutines that use the Table**

If you have user-defined VBA functions, and/or **Forms’ VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms’ VBA Subroutines in the corresponding way listed in the bullets above, depending on the specific cause(s) of the problem.

- **Table/Form pop-up menus with “SELECT \*” over the Table**

All such pop-up menus will show different fields. If this is what you want, it is OK.

Otherwise, you **fix** this by **editing all such Select operations** in the corresponding

pop-up menus.

The following is not a side effect, but I believe it is worth mentioning it:

- **Forms over the Table**

Forms over the Table will **not show the** newly added field. If this is what you wanted, then it is OK.

Otherwise, you **fix** this by **adding the field** to **each** Form (click *D.10.3.1*) based on the Table.

- **Reports over the Table**

Reports over the Table will **not show the** newly added field. If this is what you wanted, then it is OK.

Otherwise, you **fix** this by **manually adding the field** to each Report based on the Table.

Having the crashes listed above due to using a “**SELECT \***” is one of the reasons why I advise you **never** use “**SELECT \***”.

## **I.4.2 What are the side effects of deleting a Table field?**

**Deleting a Table field**<sup>78</sup> (click *B.6.1.8*) causes side effects in the following database elements:

- **Key fields of the Table, if the field is a Key field**

This side effect only happens if the field is one of the **Key** fields of the Table. If you **try** to delete any of the **Key** field(s), MS-Access will warn you that it is a **Key** field and will ask if you still want to delete the field. If you actually delete the field, this will clear the **Key** field configuration of the Table, effectively leaving the Table **without any Key field**. If this is what you want, then it is OK. However, it is quite infrequent (and risky) to have a Table without any key.

You **fix** this by **configuring the corresponding Key fields** of the Table after having deleted the field.

- **Indexes of the Table, if the field belongs to composite indexes**

This side effect only happens if the field belongs to one (or more) composite index(es) of the Table. If you **try** to delete the field and it belongs to one (or more) composite index(es), MS-Access will warn you about this, and will ask if you still want to delete the field. If you actually delete it, you will **lose all the composite indexes where the field was included**. If this is what you want, then it is OK.

You **fix** this by **configuring again the composite indexes** that you want in the Table.

- **Queries that use the field**

All **Queries** that use the field will **consider it a parameter** and will prompt the user to enter it.

You **fix** this by **editing all such Queries** and removing the field in a suitable manner.

---

<sup>78</sup> I will call “the field” to the field that has been deleted and “the Table” to the Table the field has been deleted from.

- **Queries with Insert operations inserting values into the field**

These Queries will **crash**, because the inserted value has no matching field in the target Table, because it is the Table.

You **fix** this by **editing all such Queries** and modifying the corresponding **Insert** operations removing the deleted field in a suitable manner.

- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the Table**

If you have **user-defined VBA functions**, and/or **Forms' VBA Subroutines**, with **SQL code that use the Table**, all the previous side effects will also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms' VBA Subroutines in the corresponding way listed in the bullets above.

- **Table/Form pop-up menus with Select operations including the field**

All such pop-up menus will not work.

You **fix** this by **editing all such Select operations** in the corresponding pop-up menus.

- **The Table's record validation rule, in case its expression includes the field**

MS-Access will **not** allow you to save the Table, because it detects that the expression in the record validation rule is invalid.

You **fix** this by **editing the Table's record validation rule** and suitably removing the field from it.

- **The Table's Calculated fields whose expression includes the field**

All such **Calculated** fields will show “**#Invalid**” because their corresponding expressions are now invalid and cannot be evaluated.

You **fix** this by either **adding** back the field, correcting the expression of the **Calculated** field or removing the **Calculated** field, as corresponds.

Notice that, to **prevent** this error, MS-Access will issue you the following warning message if you try to save the Table design with **Calculated** fields dependent on non-existing fields:

*“There are calculated columns in this table that depend on the column 'Col\_name'.*

*Changing or deleting this column may cause errors in one or more of the dependent calculated columns. Do you want to continue?”*

- **Relationships involving the field**

If the field is involved in one or more Relationships, deleting the field would make all such Relationships invalid. To prevent unintended errors, if the field is involved in one (or more) Relationships, MS-Access will inform you of this fact and **will not allow you to delete the field**.

If you anyway want to delete the field, you will have to **first manually edit all the Relationships** where the field is involved and either delete the Relationship itself or delete the field from the Relationship.

Even though they are not side effects, it is convenient that you are aware of the following

issues:

- **All data is lost**  
I guess it is pretty obvious, but just in case, **all the data** in the field (in **all** the records of the Table) **will be lost and cannot be recovered**.
- **Forms over the Table**  
The name of the field **will not be automatically deleted** by MS-Access. The column will remain in the Form **as an unlinked field**. If you want to **fix** this, you manually delete the column corresponding to the field from all Forms based on the Table.
- **Reports over the Table**  
The name of the field **will not be automatically deleted** by MS-Access. The column will remain in the Report **as an unlinked field**. If you want to **fix** this, you manually delete the column corresponding to the field from all Reports based on the Table.

### **I.4.3 What are the side effects of changing the name of a Table field?**

Remind that it is **not the same** to change a Table's **field name** (click *B.6.1.7*) than changing a Table's **column heading** in "**Datasheet View**" (click *H.5.1*). Changing the column heading has no side effects.

However, changing the **name** of a Table **field** causes side effects in the following database elements:

- **Queries that use the old field name**  
These Queries will **now consider it a parameter** and will request the user to input its value when the Query is run.  
You **fix** this by **editing all such Queries** and replacing the old field name by the new field name.
- **Queries with Insert operations inserting values into the old field name**  
With one exception, these Queries will now **crash**, because the old field name in the **Insert** operation will not have a matching field name in the target Table. The exception is the **Insert-one-record** operation without the list of target Table field names, that will work fine.  
You **fix** this by **editing all such Queries** and replacing in the **Insert** operations the old field name by the new field name.
- **User-defined VBA functions and Forms' VBA Subroutines that use the Table**  
If you have user-defined VBA functions, and/or **Forms' VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.  
You **fix** this by **editing the SQL code of all such** user-defined VBA functions and Forms' VBA Subroutines and modifying the corresponding SQL operations by replacing the old field name by the new field name.
- **Table/Form pop-up menus with Select operations including the old field name**  
All such pop-up menus will not work.  
You **fix** this by **editing all such Select operations** in the corresponding pop-up menus.

- **This field's validation rule**

The new field name **will not** be automatically updated in the expression of this field validation rule.

You **fix** this by **editing this field's validation rule** (click *D.5.1.5*) and replacing the old field name by the new field name.

- **This Table's record validation rule, in case it includes the old field name**

The new field name **will not** be automatically updated in the expression of this Table's record validation rule.

You **fix** this by **editing this Table's record validation rule** (click *D.8.1*) and replacing the old field name by the new field name.

Just for your information, I will now list a few cases where there are **no side effects** because in these cases MS-Access **automatically** updates the **old field name** to the **new field name**:

- **Calculated fields** in the Table that used the old field name will work fine.
- **Forms** over the Table with fields linked to the Table field will work fine.
- **Reports** over the Table with fields linked to the Table field will work fine.
- **SQL operations** involving the field name, that are used in drop-down menus, will work fine.
- **Relationships** involving the Table field name will work fine.

#### **I.4.4 What are the side effects of modifying the properties of a Table field?**

You may click:

- *"I.4.4.1 What are the side effects of changing the "Field Type" and/or "Field Size" properties of a Table field?"*
- *"I.4.4.2 What are the side effects of changing the "Required", "Allow zero length", "Validation rule" or "Indexing" properties of a Table field?"*
- *"I.4.4.3 What are the side effects of changing a drop-down menu of a Table field?"*

##### **I.4.4.1 What are the side effects of changing the "Field Type" and/or "Field Size" properties of a Table field?**

Changing the "**Field Type**" and/or "**Field Size**" properties of a Table field (click *B.6.1.7*) causes side effects in the following database elements:

- **Data in the Table field, when changing to a larger field type and/or field size**  
If you change to a new field type and/or field size **larger than** the former ones, MS-Access will do it **silently** and the field's values will be **converted** to the new field type and/or field size. I want to highlight that changing from any **numeric-like** field type to a **Short Text** field type, even if its field size is really small (e.g., 2 characters), is considered by MS-Access as changing to a **larger** field type and/or field size, and therefore, the change will be applied by MS-Access when saving the Table design, **without issuing any type of warning message**.



If you change a **numeric-like** field type to a larger field size (e.g., from **Number-Long** to **Number-Double**), the field values **may be modified** because of decimal/binary conversion rounding errors (click *G.9.3*). If you change from a **numeric-like** field type to a **Short Text** field type, the values will be **certainly modified**. First, the numeric-like values will be converted to the “**equivalent**” text string values (**Boolean** and **datetime** values will be changed to their specific text representations, while **numeric** will be changed to numeric text representations). Second, the resulting text strings will be **truncated** to the number of characters of the new value of the **Field Size** property. Most important, the resulting text string **may violate** (i.e., return *False* from) the field validation rule and/or the record validation rule, and MS-Access will not even issue a warning about it. Even if the validation rules are violated, the new values **will stay in the Table**.

- **Data in this Table field, when changing to a smaller field type and/or field size**  
If you change to a new field type and size **smaller than** the former field type and/or field size, MS-Access **cannot guarantee** that existing values will be preserved, and will show the following warning message, asking for your confirmation to proceed:

**“Some data may be lost.**

*The size of one or more fields has been changed to a shorter size. If data is lost, validation rules may be violated as a result. Do you want to continue anyway?”*

If you click on “**Yes**” and proceed to apply the change, MS-Access will do type conversion, **for all this field’s data in the Table**, from the former field type and size to the new field type and size. MS-Access will do its best to do the data conversion correctly, but **data correctness cannot be guaranteed**. Most important, the result of type conversions **may violate** (i.e., return *False* from) the field validation rule and/or the record validation rule. Even if the validation rules are violated, the new value **will stay in the Table**. You should therefore be **very sure** of what you are doing before clicking on “**Yes**”.

Some examples of data conversion that may happen are:

- If you change from a **Number-Simple** to a **Number-Long**, the former fractional values will be converted to integer (using round-half to even). If you want to know more about rounding types, you may click “*J.11.20 How do I fix a Query making rounding errors?”*”.
- If you **reduce** the “**Field Size**” property of a **Short Text** field to “**n**” characters, all stored text strings longer than “**n**” will be **truncated** to their first “**n**” characters.

Notice further that if a given type conversion **cannot** be done, the value will be **deleted**. For example, if you change from a **Number-Double** to a **Number-Simple** all the **Number-Double** values **larger than** what can be stored as a **Number-Simple** will be **deleted**, resulting in having **Null** in that field. Whenever values will be deleted, MS-Access will issue the following warning message, asking for your confirmation to proceed:

**“Microsoft Access encountered errors when converting the data.**

*The contents of fields in N record(s) were deleted.  
Do you want to proceed anyway?”*

where “*N*” is the number of records having field values that will be **deleted**, becoming **Null**.

If you click on “**Yes**”, and the values become **Null**, the result **may violate** (i.e., return **False** from) the field validation rule and/or the record validation rule as in the previous case. In this case, it may **also** violate the field properties “**Required**” and “**Allow Zero Length**”. Even if the said rules and/or properties are violated, the **Nulls will stay in the Table**.

You should therefore be **very sure** of what you are doing before clicking on “**Yes**”.

**Furthermore**, it is possible that some of the changes I have just explained on your Table data may **violate** the conditions of the Table’s **Primary Key** and/or one (or more) Table’s **indexes without duplicate values**. If this is the case, MS-Access will issue the following **additional** warning message:

*“The changes you requested to the table were not successful because they would create duplicate values in the index, primary key, or relationship. Change the data in the field or fields that contain duplicate data, remove the index, or redefine the index to permit duplicate entries and try again.”*

Regardless of you clicking “**OK**” or closing the error box, MS-Access will **remove** the **Primary** key and/or the **indexes that are violated by the converted data**, and will show the additional warning message:

*“Microsoft Access deleted n indexes on the converted fields.*

*Some data did not convert properly.”*

Notice that the **Nulls and/or duplicate values** (that resulted from the converted data) that violate the Primary key and/or indexes **will stay in the Table**.

Notice further that sometimes MS-Access **does not** remove the **Primary** key and/or the **indexes** that are violated by the converted data. In this case, MS-Access will not allow you to save the Table design, and you will have to **manually remove** the Primary key and/or indexes in order to be able to **close** the Table.

- **Queries with Union operations that use this Table field**

The new field type may affect the result of **Union** operations because MS-Access will do a type-cast among all the different field types in each column of the **Union** operation.

You **fix** this by **editing all such Queries** and modifying the corresponding **Union** operations to make them produce your intended result.

- **Queries with numeric-like SQL aggregate functions over a non-numeric-like field-type**

The numeric-like **SQL aggregate functions** **Sum()**, **Avg()**, **StDev()**, **StDevP()**, **Var()** and **VarP()** can only work over a **numeric-like** field type (click *F.7.18.7*). If you changed this field from a **numeric-like** field type to a **non-numeric-like** field type, all the Queries with a numeric-like **SQL aggregate function** over this field will crash.

You **fix** this by **editing all such Queries** and modifying them accordingly.

- **Queries and VBA functions with numeric-like domain aggregate functions over**

**a non-numeric-like field-type**

The numeric-like **domain** aggregate functions **DSum()**, **DAvg()**, **DStDev()**, **DStDevP()**, **DVar()** and **DVarP()** can only work over a **numeric-like** field (click *G.6.2* and *F.7.18.7*). If you changed this field from a numeric-like field type to a non-numeric-like field type, all the Queries and VBA functions with a numeric-like **domain** aggregate function over this field will crash.

You **fix** this by **editing all such Queries** and/or **VBA functions** and modifying them accordingly.

- **User-defined VBA functions and Forms' VBA Subroutines that use the Table**  
If you have user-defined VBA functions, and/or **Forms' VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all such** user-defined VBA functions and Forms' VBA Subroutines in the corresponding way listed in the bullets above, depending on the specific cause(s) of the problem.

- **Relationships with referential integrity involving the field**  
If the field is involved in one or more Relationships with referential integrity, changing the field type-size would make all such Relationships invalid. Remind that both fields in each master-slave pair **must** have the same "**Field Type**" and "**Field Size**".

To prevent unintended errors, when you try to change the field type-size of a field involved in one, or more, Relationships, MS-Access will inform you of this fact and **will not allow you to remove the field**.

If you want to change the field type or field size of this field, you will have to first manually edit all the Relationships where this field is involved to either remove the Relationship or remove the field from the Relationship.

- **Relationships without referential integrity involving the field**  
Master/slave field pairs of Relationships without referential integrity do not need to have the same "**Field Type**" and "**Field Size**". In spite of this, MS-Access will behave in the same way as it does for Relationships with referential integrity, as I have just described in the previous bullet point. Therefore, if you want to change the field type-size of field involved in one, or more, Relationships without referential integrity, you will have to **first manually remove all such Relationships, change the field type-size, and then manually configure again all the previously existing Relationships** Since Relationships without referential integrity are seldom used (my advice is you never use them), this problem is not very relevant.

As you have seen, changing the field type and/or the field size is an **extremely risky operation**. I advise you do not change any field type nor any field size while the database is in operation. If you really want/need to do it, my advice is that **before** doing the field type or field size change, you **backup** your MS-Access file. Then, you copy all the field values (the whole column) to an external application (typically Excel) and adapt the old values to the new field type and field size **in the specific way you want**. You may then change the field type and/or field size and paste back the new values you want in the field. This is a very risky operation, so check that all the pasted values are correct

before considering this solved. You may also want to click:

- “E.5.2 How do I paste data into MS-Access?”
- “E.5.3 How do I paste data copied from MS-Access into other applications?”

#### **I.4.4.2 What are the side effects of changing the “Required”, “Allow zero length”, “Validation rule” or “Indexing” properties of a Table field?**

Changing the properties “Required”, “Allow zero length”, field “Validation rule” or “Indexing” (click *B.6.1.7*) causes side effects in the following database elements:

- **The field value in existing Table records may violate the new property settings**  
To prevent unintended errors, MS-Access will ask you if you want that MS-Access checks if any current record violates the new settings. If click “Yes”, in case any record violates the new properties, MS-Access will ask you if you want to keep the new properties or you want to go back to the old properties: you just need to answer that you want to keep the new properties. This will keep the new properties but will leave the records that violate them unchanged. Having records that violate field properties will most likely cause database errors quite difficult to detect. I strongly advise you avoid having Table records that violate the field properties.

You **fix** the records that violate the field properties by **copying all** the field values (the whole column) to an external application (typically Excel) and adapting the old values to comply with the new field properties **in the specific way you want**. Then paste back the new values into the field. Remind that bulk copy/paste operations are risky: before doing them, I suggest you click:

- “E.5.2 How do I paste data into MS-Access?”
- “E.5.3 How do I paste data copied from MS-Access into other applications?”

#### **I.4.4.3 What are the side effects of changing a drop-down menu of a Table field?**

If the drop-down menu does not have the property “Limit to List=Yes”, this will not cause any side effect.

If the drop-down menu has the property “Limit to List=Yes”, this will not restrict the values that you can enter into this field to the ones of the drop-down menu. This is most likely what you want, but I wanted to point it out just in case.

If you want to know more about drop-down menus, you may click *K.1.8*.

#### **I.4.5 What are the side effects of changing the order of Table fields?**

Remind that it is **not the same** to change the order of Table fields (click *H.3*) than changing the order of columns of a Table in “**Datasheet View**” (click *H.3.1*).

Changing the order of a columns of a Table in “**Datasheet View**” has no side effects.

Changing the order of Table fields causes side effects if you are using “**SELECT \***” over the Table, you have Forms over the Table or you have **Insert** operations into the Table. Side effects will affect the following database elements:

- **Queries with “SELECT \*” over the Table in a Union operation**  
These Queries will **produce wrong results** because the “**SELECT \***” will produce

the fields in a different order, thus wrongly combining the values with the other **input** record-list to the **Union** operation.

You **fix** this by **editing all such Queries** and either:

- Replace the “\*” by an explicit list of the corresponding Table fields in the proper order.
- Modify all the other **input** record-lists of the **Union** operation to match the new field order.

- **Queries with Insert-one-record operation without the optional list of target Table field names that have the Table as its target Table**

These Queries will insert wrong values, because the order of values was aligned with the old order of fields, and therefore, will not be aligned with the new order of fields.

You **fix** this by **editing all such Queries** and modify the corresponding **Insert** operations. If you want to know more about **Insert** operations, you may click “[F.13.2 What is an Insert operation and how do I write it?](#)”.

- **User-defined VBA functions and Forms’ VBA Subroutines that use the Table**

If you have user-defined VBA functions, and/or **Forms’ VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all such** user-defined VBA functions and Forms’ VBA Subroutines in the corresponding way listed in the bullets above, depending on the specific cause(s) of the problem.

The following are not side effects, but I believe it is worth mentioning them:

- **Forms over the Table**

Forms over the Table will **not show** the same field order as the one that has been configured in the Table. This is not a problem at all, but you should be aware of it. In case you do want to change the field order of Forms over this Table, you should manually change it for each such Form.

- **Reports over the Table**

Reports over the Table will **not show** the same field order as the one that has been configured in the Table. This is not a problem at all, but you should be aware of it. In case you do want to change the field order of Reports over this Table, you should manually change it for each such Report.

## **I.5 What are the side effects of modifying my Tables?**

You may click:

- “[I.5.1 What are the side effects of adding a new Table?](#)”
- “[I.5.2 What are the side effects of deleting a Table?](#)”
- “[I.5.3 What are the side effects of changing the name of a Table?](#)”
- “[I.5.4 What are the side effects of modifying a Table field?](#)”
- “[I.5.5 What are the side effects of modifying the order of fields in the Table?](#)”
- “[I.5.6 What are the side effects of modifying the indexes or the Key fields of a Table?](#)”

- “I.5.7 What are the side effects of modifying the record validation rule of a Table?”

### **I.5.1 What are the side effects of adding a new Table?**

Adding new Tables (click *D.3*) does not have any side effect.

### **I.5.2 What are the side effects of deleting a Table?**

Deleting a Table (click *B.4.1.13*) causes side effects in the following database elements:

- **Queries using the Table**  
These Queries will now **crash**.  
  
You **fix** this by **editing all such Queries** and removing the references made to the deleted Table in a suitable manner.
- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the Table**  
If you have **user-defined VBA functions**, and/or **Forms' VBA Subroutines**, with **SQL code that use the Table**, the side effects in the previous bullet point will also affect them in exactly the same way.  
  
You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms' VBA Subroutines in the corresponding way listed in the bullet above.
- **Table/Form pop-up menus with Select operations including the deleted Table**  
All such pop-up menus will not work.  
  
You **fix** this by **edit all such Select operations** in the corresponding pop-up menus.
- **Forms over the deleted Table**  
These Forms will stop working. When you open any of them, you will get an informative message indicating that the origin of data does not exist.  
  
You **fix** this by **either manually deleting** each of the orphan Forms, **or by manually linking** each of them to an existing or newly created Table.
- **Reports over the deleted Table**  
These Reports will stop working. When you open any of them, you will get an informative message indicating the origin of data does not exist.  
  
You **fix** this by **either manually deleting** each of the orphan Reports, **or by manually linking** each of them to an existing or newly created Table.
- **Relationships involving the Table**  
If the Table is involved in one or more Relationships, deleting the Table would make all such Relationships invalid. To prevent unintended errors, when you try to remove a Table involved in one, or more, Relationships, MS-Access will inform you of this fact and will ask if you want MS-Access to remove all such Relationships. If you click on “**Yes**”, MS-Access will remove all such Relationships and will also remove the Table.

Even though this is not a side effect, it is convenient that you are aware of the following issue:

- I guess it is pretty obvious, but just in case, **all the data** in the deleted Table **will be lost and cannot be recovered**.

### **I.5.3 What are the side effects of changing the name of a Table?**

If you change the name of a Table (click *B.4.1.8*), MS-Access will automatically update the new Table name in all Relationships and Forms. Therefore, Relationships and Forms suffer no side effects from changing a Table name.

However, changing a Table name will causes side effects in the following database elements:

- **Queries using the Table**

These Queries will now **crash**.

You **fix** this by **editing all such Queries** and replacing the old Table name by the new Table name.

- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the Table**

If you have **user-defined VBA functions**, and/or **Forms' VBA Subroutines**, with **SQL code that use the Table**, the side effects in the previous bullet point will also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms' VBA Subroutines in the corresponding way listed in the bullet above.

- **Table/Form pop-up menus with Select operations including the Table**

All such pop-up menus will not work.

You **fix** this by **editing all such Select operations** in the corresponding pop-up menus replacing the old Table name by the new Table name.

### **I.5.4 What are the side effects of modifying a Table field?**

This is quite complex and has a full chapter on its own, you may click:

- *"I.4 What are the side effects of modifying my Table fields?"*

### **I.5.5 What are the side effects of modifying the order of fields in the Table?**

Remind that it is **not the same** to change the order of Table fields (click *B.6.1.8*) than changing the order of Table columns shown in "**Datasheet View**" (click *H.3*).

Changing the order of Table columns in "**Datasheet View**" has no side effects.

In respect to the side effects of **changing the order of fields** in the Table, you may click *"I.4.5 What are the side effects of changing the order of Table fields?"*.

### **I.5.6 What are the side effects of modifying the indexes or the Key fields of a Table?**

You may click:

- *"I.5.6.1 What are the side effects of changing an index with duplicate values?"*
- *"I.5.6.2 What are the side effects of adding an index without duplicate values?"*
- *"I.5.6.3 What are the side effects of changing an index without duplicate values?"*
- *"I.5.6.4 What are the side effects of adding or changing the Key fields of a Table?"*



### **I.5.6.1 What are the side effects of changing an index with duplicate values?**

**Adding or removing** an index **with** duplicate values causes **no side effects**.

**Adding or removing fields** to/from an index **with** duplicate values causes **no side effects**.

Even though this is not a side effect, it is convenient that you are aware that **removing** indexes typically degrades the performance of your database.

**Removing** an index **with** duplicate values will most likely degrade the performance of your database.

### **I.5.6.2 What are the side effects of adding an index without duplicate values?**

**Adding** an index **without** duplicate values, or **changing** the properties of an existing index from being **with** duplicate values to become **without** duplicate values, causes side effects in the following database elements:

- **Relationships with referential integrity with exactly the slave fields of this index**  
If the index fields are part of the **slave** fields of one or more Relationships **with referential integrity**, configuring the index as **without** duplicate values, would **change** all such Relationships from being **one-to-many** to **one-to-one**.

- **Newly input Table data**

MS-Access will prevent records with duplicate values in the index fields. This is most likely what you want, but just in case I wanted to point it out.

To prevent unintended errors, when you **try to do add the index**, MS-Access will check if the Table contains any record with duplicate values over the index fields: in case there is on (or more) such records, MS-Access **will not allow you** to do add the index (click *L.2.6*).

### **I.5.6.3 What are the side effects of changing an index without duplicate values?**

**Removing** an index **without** duplicate values, or **adding/removing** fields to/from the index, or **changing** the index from being **without** duplicate values to become **with** duplicate values will cause side effects in the following database elements:

- **Relationships with referential integrity with exactly the master fields of this index**

If **exactly** the index fields are the **master** fields of one or more Relationships **with referential integrity**, the changes indicated above will make the Relationships invalid. Remind that the master fields in a Relationship with referential integrity **must** always have an associated index without duplicate values and without **Nulls**.

To prevent unintended errors, when you **try to do any** of the changes above, MS-Access will inform you that the index is used in a Relationship and **will not allow you** to do the change (click *L.2.9* for an index or *L.2.8* for the Key fields).

If you **want to do** any of the changes above, you will have to either **manually** remove all the affected Relationships or to **manually** configure them as without referential integrity. Notice that changing your Relationships does not have side effects, but you have to be **very sure** that you want that.

Notice also that if you **try to remove field(s)** from an index without duplicate values,

MS-Access will check if the Table contains any record with duplicate values over the remaining index fields: in case there is on (or more) such records, MS-Access **will not allow you** to remove the field(s) from the index (click *L.2.6*).

**Removing** an index **without** duplicate values will most likely degrade the performance of your database.

#### **I.5.6.4 What are the side effects of adding or changing the Key fields of a Table?**

The **Key** fields of a Table **must** have an associated index **without duplicate values** and without **Nulls**. For this reason, changing the **Key** fields of a Table will cause the same side effects as changing the fields of an index **without duplicate values**, as described in:

- “*I.5.6.2 What are the side effects of adding an index without duplicate values?*”
- “*I.5.6.3 What are the side effects of changing an index without duplicate values?*”

#### **I.5.7 What are the side effects of modifying the record validation rule of a Table?**

**Deleting** the record validation rule (click *B.6.4*) causes no side effect.

**Creating** or **modifying** (click *D.8.1*) the record validation rule causes side effects in the following database elements:

- **Existing Table data**  
MS-Access will check the currently existing data in the Table, and in case it is not compliant with the new record validation rule, will issue a warning. However, the data will **not** be modified, and the record validation rule will be inconsistent with existing Table data.

You **fix** this by **copying all** the data in the fields used in the record validation rule to an external application (typically Excel), modifying it to be compliant with the new record validation rule, and pasting back into the Table. Remind that bulk copy/paste operations are risky. Before doing them, I suggest you click:

- “*E.5.2 How do I paste data into MS-Access?*”
- “*E.5.3 How do I paste data copied from MS-Access into other applications?*”

### **I.6 What are the side effects of modifying my Queries?**

You may click:

- “*I.6.1 What are the side effects of adding a new Query?*”
- “*I.6.2 What are the side effects of deleting a Query?*”
- “*I.6.3 What are the side effects of modifying the functionality of a Query?*”
- “*I.6.4 What are the side effects of changing the name of a Query?*”
- “*I.6.5 What are the side effects of adding a new field to a Query?*”
- “*I.6.6 What are the side effects of deleting a Query field?*”

- “I.6.7 What are the side effects of changing the name of a Query field?”
- “I.6.8 What are the side effects of changing the data type of a Query field?”
- “I.6.9 What are the side effects of modifying the order of fields in a Query?”

### **I.6.1 What are the side effects of adding a new Query?**

Adding new Queries (click F.4.5) does not have any side effect.

### **I.6.2 What are the side effects of deleting a Query?**

Deleting a Query causes side effects in the following database elements:

- **Queries using the Query**  
These Queries will now **crash**.  
  
You **fix** this by **editing all such Queries** and removing the references made to the deleted Query in a suitable manner.
- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the Query**  
If you have **user-defined VBA functions**, and/or **Forms' VBA Subroutines**, with **SQL code that use the deleted Query**, the side effect in the previous bullet point will also affect them in exactly the same way.  
  
You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms' VBA Subroutines in the corresponding way listed in the previous bullet point.
- **Table/Form pop-up menus with Select operations including the deleted Query**  
All such pop-up menus will not work.  
  
You **fix** this by **editing all such Select operations** in the corresponding pop-up menus.
- **Table/Form pop-up menus with the deleted Query**  
All such pop-up menus will not work.  
  
You **fix** this by **editing all such pop-up menus** in a suitable manner.
- **Forms over the deleted Query**  
These Forms will stop working. When you open any of them, you will get an informative message indicating the origin of data does not exist.  
  
You **fix** this by **either manually deleting** each of the orphan Forms, **or by manually linking** each of them to an existing or newly created Query.
- **Reports over the deleted Query**  
These Reports will stop working. When you open any of them, you will get an informative message indicating the origin of data does not exist.  
  
You **fix** this by **either manually deleting** each of the orphan Reports, **or by manually linking** each of them to an existing or newly created Query.

### **I.6.3 What are the side effects of modifying the functionality of a Query?**

If you change the functionality of the Query, and it therefore produces now different

results, this may cause side effects in the following database elements:

- **Queries using the Query**  
These Queries may now produce different results.
- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the Query**  
If you have **user-defined VBA functions**, and/or **Forms' VBA Subroutines**, with **SQL code that use the deleted Query**, the side effect in the previous bullet point will also affect them in exactly the same way.
- **Table/Form pop-up menus with Select operations including the Query**  
All such pop-up menus will now work differently or will not work.
- **Table/Form pop-up menus with the Query**  
All such pop-up menus will now work differently or will not work.

You **fix** this by **creating a copy** of the former Query and either:

- Keep the same Query name in the copy of the former Query and assign a new Query name to the Query with modified functionality.
- Assign a new Query name to the copy of the former Query and replace its former name by the new name in all the places where it was used from the three bullet points above.

#### **I.6.4 What are the side effects of changing the name of a Query?**

Changing a Query name (click *B.4.1.8*) will cause side effects in the following database elements:

- **Queries using the Query**  
These Queries will now **crash**.  
  
You **fix** this by **editing all such Queries** and replacing the old Query name by the new Query name.
- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the Query**  
If you have **user-defined VBA functions**, and/or **Forms' VBA Subroutines**, with **SQL code that use the deleted Query**, the side effect in the previous bullet point will also affect them in exactly the same way.  
  
You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms' VBA Subroutines in the corresponding way listed in the previous bullet point.
- **Table/Form pop-up menus with Select operations including the Query**  
All such pop-up menus will not work.  
  
You **fix** this by **editing all such Select operations** in the corresponding pop-up menus and replacing the old Query name by the new Query name.
- **Table/Form pop-up menus with the Query**  
All such pop-up menus will not work.  
  
You **fix** this by **editing all such pop-up menus** and replacing the old Query name by the new Query name.

### I.6.5 What are the side effects of adding a new field to a Query?

Adding a new field to a Query causes side effects if you are using “**SELECT \***” over the Query or you have Forms over the Table. Side effects will affect the following database elements:

- **Queries with “**SELECT \***” over the Query**

The result of the **Select** operation will show an additional field. This is what you may want in some cases. However, this additional field may propagate through other “**SELECT \***” higher up the SQL operation hierarchy and cause some of the problems listed in the following bullet points.

You **fix** this by **editing all such Queries** and replacing the “**\***” by an explicit list of the corresponding Query fields wherever it is needed: see the problem description in the following bullet points.

- **Queries with “**SELECT \***” over the Query in a Union operation**

The Queries with “**SELECT \***” will **crash** because the “**SELECT \***” will include the newly added Query field in its **output** fields, thus having a **different number** of fields than **the other input** record-list of the **Union** operator. Remind that both **input** record-lists to a **Union** operator **must** have the same number of fields.

You **fix** this by **editing all such Queries** and either:

- Replace the “**\***” by an explicit list of the corresponding Query fields.
- Modify all the other **input** record-lists of the **Union** operations to match the newly added field.

- **Queries with “**SELECT \***” over the Query, and the **Select** is used in an **Insert-many-records** operation**

The Queries will (most likely) **crash**. In the case of **using** the optional list of target Table field names, the **number** of fields will now **not** be the same. In case of **omitting** the optional list of target Table field names, the newly added Query field most likely does not exist in the target Table, and this **crashes** the said Queries. In case the newly added field exists in the target Table, this is most likely an unintended coincidence, and the **Insert operation** will insert wrong values in that field: this error may be quite difficult to detect.

You **fix** this by **editing all such Queries** and either:

- Replace the “**\***” by an explicit list with the corresponding Query fields.
- Modify the target Table to also include the newly added Query field (in case this is coherent with your database design).

If you want to know more about **Insert** operations, you may click “*F.13.2 What is an Insert operation and how do I write it?*”.

- **User-defined VBA functions and Forms’ VBA Subroutines that use the Query**

If you have user-defined VBA functions, and/or **Forms’ VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms’ VBA Subroutines in the corresponding way listed in the bullets above, depending on the specific cause(s) of the problem.

- **Table/Form pop-up menus with “SELECT \*” over the Query**  
All such pop-up menus will show different fields. If this is what you want, it is OK.  
Otherwise, you **fix** this by **editing all such Select operations** in the corresponding pop-up menus.

The following is not a side effect, but I believe it is worth mentioning it:

- **Forms over the Query**  
Forms over the Query will **not show** the newly added field. If this is what you wanted, then it is OK.  
Otherwise, you **fix** this by **adding** this field to each Form based on the Query where you want the field to appear.
- **Reports over the Query**  
Reports over the Query will **not show** the newly added field. If this is what you wanted, then it is OK.  
Otherwise, you **fix** this by **manually adding** this field to each Report based on the Query where you want the field to appear.

Having the crashes listed above due to using a “SELECT \*” is one of the reasons why I advise you **never** use “SELECT \*”.

### **I.6.6 What are the side effects of deleting a Query field?**

Deleting a **Query field** causes side effects in the following database elements:

- **Queries that use the deleted field name**  
All **Queries** that use the deleted field name will **consider it a parameter** and will prompt the user to enter it.  
You **fix** this by **editing all such Queries** and removing the deleted field in a suitable manner.
- **User-defined VBA functions and Forms’ VBA Subroutines with SQL code that use the Query**  
If you have **user-defined VBA functions**, and/or **Forms’ VBA Subroutines**, with **SQL code that use the Query**, side effects in the previous bullet point will also affect them in exactly the same way.  
You **fix** this by **editing the SQL code of all** such user-defined VBA functions and Forms’ VBA Subroutines in the way listed in the bullet above.
- **Table/Form pop-up menus with Select operations including the deleted field name**  
All such pop-up menus will not work.  
You **fix** this by **editing all such Select operations** in the corresponding pop-up menus.

Even though they are not side effects, it is convenient that you are aware of the following issues:

- **Forms over the Query**  
The deleted field name **will not be automatically deleted** by MS-Access. The column will remain in the Form **with unlinked field values**. If you want to **fix** this,

you manually delete the column from all Forms based on the Query.

- **Reports over the Query**

The deleted field name **will not be automatically deleted** by MS-Access. The field will remain in the Report **with unlinked field values**. If you want to **fix** this, you manually delete the field from all Reports based on the Query.

### **I.6.7 What are the side effects of changing the name of a Query field?**

Remind that it is **not the same** to change a Query's **field name** (click *H.5*) than changing the Query's **column heading** in "**Datasheet View**" (click *H.5.2*). Changing the column heading in "**Datasheet View**" has no side effects.

However, changing the **name** of a Query **field** causes side effects in the following database elements:

- **Queries that use the old field name**

These Queries will **now consider it a parameter** and will request the user to input its value when the Query is run.

You **fix** this by **editing all such Queries** replacing the old field name by the new field name.

- **User-defined VBA functions and Forms' VBA Subroutines that use the Query**

If you have user-defined VBA functions, and/or **Forms' VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all such** user-defined VBA functions and Forms' VBA Subroutines modifying the corresponding SQL operations by replacing the old field name by the new field name.

- **Table/Form pop-up menus with Select operations including the old field name**

All such pop-up menus will not work.

You **fix** this by **editing all such Select operations** in the corresponding pop-up menus.

Just for your information, I will now list a few cases where there are **no side effects** because in these cases MS-Access **automatically** updates the **old field name** to the **new field name**:

- **Forms** over the Query with fields linked to the Query field will work fine.
- **Reports** over the Query with fields linked to the Query field will work fine.
- **SQL operations** involving the field name, that are used in drop-down menus, will work fine.

### **I.6.8 What are the side effects of changing the data type of a Query field?**

Changing the data type of a Query field causes side effects in the following database elements:

- **Queries with Union operations that use this Query field**

The new field type may affect the result of a **Union** operations because MS-Access

will do a type-cast among all the different field types in each column of the **Union** operation.

You **fix** this by **editing all such Queries** and modifying the corresponding **Union** operations to make them produce your intended result.

- **Queries with numeric-like aggregate functions over a non-numeric-like field-type**

The **numeric-like SQL aggregate functions** “**Sum ()**”, “**Avg ()**”, “**StDev ()**”, “**StDevP ()**”, “**Var ()**” and “**VarP ()**” can only work over a field with a numeric-like data type (click *F.7.18.7*). The same happens with the equivalent **numeric-like domain aggregate functions** (click *G.6.2* and *F.7.18.7*). If you changed this field from a numeric-like data type to a non-numeric-like data type, all the Queries with a numeric-like aggregate function over this field will crash.

You **fix** this by **editing all such Queries** and modifying them accordingly.

- **User-defined VBA functions and Forms’ VBA Subroutines that use the Query**  
If you have user-defined VBA functions, and/or **Forms’ VBA Subroutines**, with **SQL code**, all the previous side effects may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all such** user-defined VBA functions and Forms’ VBA Subroutines in the corresponding way listed in the bullets above, depending on the specific cause(s) of the problem.

### **I.6.9 What are the side effects of modifying the order of fields in a Query?**

Remind that it is **not the same** (click *H.3*) to change the order of Query fields than changing the order of columns of a Query in “**Datasheet View**” (click *H.3.1*).

Changing the order of a columns of a Query in “**Datasheet View**” has no side effects.

However, changing the order of a Query field causes side effects in the following database elements:

- **Queries with “SELECT \*” over the Query in a Union operation**  
These Queries will **produce wrong results** because the “**SELECT \***” will produce the fields in a different order, thus wrongly combining the values with the other input record-list to the **Union** operation.

You **fix** this by **editing all such Queries** and either:

- Replace the “**\***” by an explicit list of the corresponding Query fields in the proper order.
- Modify all the other input-lists of the **Union** operation to match the new field order.

- **User-defined VBA functions and Forms’ VBA Subroutines that use the Query**  
If you have user-defined VBA functions, and/or **Forms’ VBA Subroutines**, with **SQL code**, the side effects in the previous bullet point may also affect them in exactly the same way.

You **fix** this by **editing the SQL code of all such** user-defined VBA functions and Forms’ VBA Subroutines in the corresponding way listed in the previous bullet point.



The following are not side effects, but I believe it is worth mentioning them:

- **Forms over the Query**  
Forms over the Query will **not show** the same field order as the one that has been configured in the Query. This is not a problem at all, but you should be aware of it. In case you do want to change the field order of Forms over this Query, you should manually change it for each such Form.
- **Reports over the Query**  
Reports over the Query will **not show** the same field order as the one that has been configured in the Query. This is not a problem at all, but you should be aware of it. In case you do want to change the field order of Reports over this Query, you should manually change it for each such Report.

## **I.7 What are the side effects of modifying my user-defined VBA functions?**

**Adding** user-defined VBA functions **does not cause** any side effect.

**Modifying the internal computations of** a user-defined VBA function **does not cause** any side effects, but it will obviously cause direct effects, anywhere that is used. These direct effects are most likely what you want, but I am pointing this out just in case.

**Deleting** a user-defined VBA function or **adding/removing** its arguments causes side effects in the following database elements:

- **Queries using the function**  
These Queries will now **crash**.  
  
You **fix** this by **editing all such Queries** and removing/modifying the references made to the deleted/modified function in a suitable manner.
- **User-defined VBA functions and Forms' VBA Subroutines with SQL code that use the function**  
These functions and Subroutines will now **crash**.  
  
You **fix** this by **editing all such functions and Subroutines** and removing/modifying the references made to the deleted/modified function in a suitable manner.
- **Forms using the function**  
These Forms will now **crash**.  
  
You **fix** this by **editing all such Forms** and removing/modifying the references made to the deleted/modified function in a suitable manner.
- **Reports using the function**  
These Reports will now **crash**.  
  
You **fix** this by **editing all such Reports** and removing/modifying the references made to the deleted/modified function in a suitable manner.

**Replacing** the function data type by a new one may cause the same side effects above if the new data type is not equivalent to the former one.

## **I.8 What are the side effects of modifying my Relationships, Forms and/or Reports?**

Modifying your Relationships, Forms and/or Reports does **not** cause any side effect.

You can add/delete/modify your Relationships, Forms and/or Reports, and this will only cause the **direct effects** of the change you did, which most likely is **exactly** what you wanted.

## PART J. DEBUGGING MY SQL QUERIES

When you want to fix a bug (error or crash) in a Query, you have two very different cases:

- The bug is in **test-and-proven** Query
- The bug is in a **non-test-and-proven** Query

The **first case** applies when the Query has already been in operation for a substantial time (e.g., months) without any apparent problem. If you get a bug in such a Query, click:

- [\*“J.1 How do I fix an error/crash in a test-and-proven Query?”\*](#)

The **second case** applies when you are now writing the Query, or when it has been in operation for a short time (e.g., a few weeks or less). If you get a bug in such a Query, click:

- [\*“J.2 How do I fix an error/crash in a non-test-and-proven Query?”\*](#)

If you want more information about debugging, you may click:

- [\*“J.3 How do I debug by commenting/uncommenting?”\*](#)
- [\*“J.4 How do I debug in progressive steps?”\*](#)
- [\*“J.5 How do I debug inside out?”\*](#)
- [\*“J.6 How do I debug my same-level code linearly?”\*](#)
- [\*“J.7 How do I debug the current uncommented SQL operation at each step?”\*](#)
- [\*“J.8 How do I fix a syntax error that prevents saving a Query?”\*](#)
- [\*“J.9 How do I fix a crash from a syntax error?”\*](#)
- [\*“J.10 How do I fix a crash from a run-time error?”\*](#)
- [\*“J.11 How do I fix defective Query results?”\*](#)
- [\*“J.12 What do I do when I just cannot fix a Query?”\*](#)
- [\*“J.13 Why should I always compare the results of an existing Query?”\*](#)
- [\*“J.14 What Null-related bugs can I get?”\*](#)
- [\*“J.15 What exception-value bugs can I get?”\*](#)
- [\*“J.16 What data type bugs can I get?”\*](#)

### J.1 How do I fix an error/crash in a test-and-proven Query?


If the error/crash is in a **non-test-and-proven** Query, click [\*“J.2 How do I fix an error/crash in a non-test-and-proven Query?”\*](#).

If the error/crash is in a **test-and-proven** Query, it is **most likely** because of a **transient** software problem or because of **data incoherence** (click *L.1*), and **not** because of a bug in the Query.

Therefore, you should **never** rush to debug the Query, and **rather** you should **first** take

the following **sequential** actions:

### 1. Retry, as you usually do with most software programs:

- a) Retry a couple times.
- b) Click on the **Compact and Repair Database** “” icon from the “**Database Tools**” Ribbon and try again.
- c) Close and open MS-Access and try again.

If the Query error/crash stays the same, go to the next step (number 2).

### 2. Run the Query that checks the correctness of your data:

Data inconsistency is by far **the most frequent cause** when you get an error after the database has been tested and has already been in operation for some time. Even if you have sound validation rules in your fields and Tables, it is very likely that some erroneous value combinations produce Query/function crashes and/or wrong Query results.

If the Query detects any data inconsistencies, fix them and try again.

If you want to know more about a Query that verifies the correctness of your data, you may click “*K.6.11 How do I design a data check Query?*”.

If the Query error/crash stays the same, go to the next step (number 3).

### 3. Remind what record changes you have done recently:

Even if your Query to detect the correctness of your data is very good and complete, there can be errors it does not detect. Therefore, think what record changes you have done recently:

- Have you copied/pasted bulk records?
- Have you corrected the values of some records?
- Have you changed the value of a field that is a Relationship master field?
- Have you changed anything in the database design?
- What are the latest records you have introduced?

Then, check the correctness of the said changes.

In case you detect any error in the data, fix it **and also** consider if it is worth enhancing your Query to detect the correctness of data including a check that would have automatically detected it.

If the Query error/crash stays the same, go to the next step (number 4).

### 4. Restart the computer and try again:

If the Query error/crash stays the same, go to the next step (number 5).

### 5. Think if you have modified any auxiliary Query:

Recall if you have recently modified any Query that is invoked (directly or indirectly) in the Query that is failing. If this is the case, then go and debug **that** Query that you have recently modified. To do it, you may click “*J.2 How do I fix an error/crash in a non-test-and-proven Query?*”.

If you have not recently modified any auxiliary Query, go to the next step (number 6).

### **6. Debug your Query code:**

If you have done all the previous steps, then the error/crash may actually be in the Query code. You should therefore debug the Query clicking “*J.2 How do I fix an error/crash in a non-test-and-proven Query?*”. When you debug the Query, rule out all syntax errors, because the Query was running, and therefore it cannot have a syntax error.

Even though your Query code has been working well for months, it is still possible that there is an error that only happens under infrequent combination of correct values.

Finally, if you want a brief explanation on why can you get an error/crash in a test-and-proven Query, you may click “*L.1 Why can I get an error/crash in a test-and-proven database?*”.

## **J.2 How do I fix an error/crash in a non-test-and-proven Query?**

If the error/crash is in a **test-and-proven** Query, click “*J.1 How do I fix an error/crash in a test-and-proven Query?*”.

If the error/crash is in a **non-test-and-proven** SQL Query, debug it following these principles:

- **Comment/uncomment your SQL code.**
- **Debug your SQL code in progressive steps.**
- **Debug your SQL code inside out.**
- **Debug your same-level SQL code linearly.**
- **Debug the SQL operation at each step.**
- **If you get stuck, do a workaround.**
- **Always compare the results of an existing Query.**

If you want more detail about these debugging principles above, you may click:

- “*J.3 How do I debug by commenting/uncommenting?*”
- “*J.4 How do I debug in progressive steps?*”
- “*J.5 How do debug inside out?*”
- “*J.6 How do I debug my same-level code linearly?*”
- “*J.7 How do I debug the current uncommented SQL operation at each step?*”
- “*J.12 What do I do when I just cannot fix a Query?*”
- “*J.13 Why should I always compare the results of an existing Query?*”

## **J.3 How do I debug by commenting/uncommenting?**

You **debug** your SQL Query code by progressively “**commenting**” and “**uncommenting**” different **fragments** of it.

**Commenting** is done by **selecting a number of lines of SQL code** and **prefixing each line with two consecutive hyphens “--”**. This converts these **lines of code into SQL comments**, so they are **ignored** by the SQL interpreter.

**Uncommenting** is done by **selecting a number of commented lines of SQL code** and **removing** their initial **two consecutive hyphens “--”** that had been previously added. Removing the two-hyphen prefix from each line makes that SQL code **effective** again, being processed by the SQL interpreter.

You most frequently **comment/uncomment complete lines** of SQL code, and not just part of the lines. You do this by using the comment/uncomment function of the plug-in “**Access SQL Editor**” (click *F.5.5.1*).

You **leave uncommented** one specific fragment of code whose syntax and functionality you want to **check**. The fragment that you have left uncommented **must be an executable SQL operation** (i.e., a **Select, Union** or **Transform**), so you can actually run it and check its results. For this reason, I will call this uncommented fragment the “**current uncommented SQL operation**”.

Notice that the uncommented lines that compose the **current uncommented SQL operation** of your SQL code **are not necessarily consecutive**.

Notice also that on a number of cases commenting lines **does not** result in a **correct SQL operation**. If this happens, you will have to **manually** adjust the code (e.g., adding parenthesis, commenting part of a line of code, ...) to get a **correct SQL operation**. This depends on how you have formatted your SQL code. This is clearly undesirable, because when doing this editing you may introduce further errors. This is why it is very important to **format** your SQL code in a way that minimizes the need for **manual** adjustments. If you want to know more about this, you may check the bullet point about parentheses clicking on “*K.4.3 How do I write readable (maintainable) SQL Queries?*”.

## **J.4 How do I debug in progressive steps?**

You debug your SQL code in **progressive steps** because it is the best way to **locate** and **fix** the error(s) that the Query contains.

At **each debugging step**, you leave a **different uncommented SQL operation** (click *J.3*) of your Query code (called the “**current uncommented SQL operation**”) where you will focus your attention. Analyzing at each step **only one SQL operation** of your code, and checking the results it produces, makes it easier to **locate** and **fix** the possible **errors** in that specific **SQL operation**.

By **progressively** focusing on **only one SQL operation** of your code, and checking the results it produces, you can **orderly debug, in a number of steps**, the **complete SQL code** of your Query.

This is a very effective way of **finding** and **fixing** all your Query **bugs**.

You may click on the following two chapters to know how to **select the current uncommented SQL operation at each progressive debugging step**:

- “*J.5 How do debug inside out?*”
- “*J.6 How do I debug my same-level code linearly?*”

You may click on the following chapter to know how to **debug the current**

**uncommented SQL operation at a given step:**

- “*J.7 How do I debug the current uncommented SQL operation at each step?*”

## **J.5 How do debug inside out?**

SQL Query code is debugged **inside out**. This means that you first **comment** (click *J.3*) the **outer** SQL code in the Query, and you leave **uncommented one inner SQL operation**. You debug this **inner** SQL operation as I indicate “*J.7 How do I debug the current uncommented SQL operation at each step?*”. Once it is correct, you uncomment a **more exterior** SQL operation that encloses this one, and you debug it with the same procedure.

When you are not experienced, my advice is if you debug the SQL operations one by one, starting in the innermost ones, and progressively going upwards until you reach the complete Query. When you are more experienced, you can debug by focusing on larger inner SQL operations (not necessarily the innermost ones), and going outwards several steps at a time, and not following one-by-one the next exterior SQL operation to the last one debugged.

Notice that the structure of SQL code is like a tree, where **the** outermost SQL operation is the root and **each** of the **several** innermost SQL operations are the final branches of the tree. The tree **nodes** are either **Union** operations or **Join** operations.

If you want more information about this, you may click:

- “*J.6 How do I debug my same-level code linearly?*”
- “*J.7 How do I debug the current uncommented SQL operation at each step?*”

## **J.6 How do I debug my same-level code linearly?**

When you are **progressively** debugging a Query **inside-out**, at a **given step** you may have several **Select** operations **at the same level**. These **Select** operations at the same level may either be:

- **Several Select** operations bound by one or more **Union** operator(s).
- **Two Select** operations bound by a **Join** operator<sup>79</sup>.

You debug these two cases as I now indicate.

### **Several same level Selects bound by Union operators**

Debug them **linearly**. This implies commenting (click *J.3*) all the **Union** operations, **except one Select operation** (or a few consecutive **Select** operations if each of them is very simple). This allows you to focus on a **current uncommented SQL operation** composed of one **Select** operation (or a few of them composed with **Union** operators). You **run the current uncommented SQL operation**, and check if its results are correct. If results are wrong, you **fix** the current uncommented SQL operation code until it is correct. Once it is correct, you uncomment the **next Select** operation in the associative **Union** operations, until you have debugged the whole block of **Select** operations.

---

<sup>79</sup> In case it is a **Cross-Join**, it can be **more than two Select** operations, because it is associative. You can anyway debug this case as I indicate for the case of other **Joins** with only two **Select** operations.

If you want to save time from commenting/uncommenting, you can debug the associative **Union** operations in an **incremental** way. This means that you do not comment again the **former** “current uncommented SQL operations” that were run and **worked well**. In this way, you have a progressively larger uncommented current uncommented SQL operation, until you have fixed all the errors and what remains uncommented is the whole block of associative **Union** operations.

An alternative of commenting and debugging the **Select** operations one by one (or few by few) is doing something similar to a **binary search**. This is particularly useful when you are debugging a Query crash, or a Query syntax error, where the effect of the error is trivial to notice. To do such a binary search, you comment **approximately half** of the complete block of associative **Union** operations. You run it, and if the Query works, the error (crash or syntax error) is **in the other half**. If the Query is still erroneous, the error is **in this half**. You repeat the process with the half that contains the error, over and over, until you find and fix the error. With this **binary-search** approach you can find the error in a fairly quick way.

### Two same level Selects bound by a Join operator

You debug **each** of the **left** and **right input** SQL operations, and then debug the **Select** operation that is enclosing the **Join** operation, as follows:

1. **Comment** (click *J.3*) the clauses of the enclosing **Select** operation and of the **Join** operation. This is, comment the “**SELECT**”, “**FROM**”, “**WHERE**”, “**GROUP BY**”, “**HAVING**” and “**ORDER BY**” clauses of the enclosing Select operation and also **comment** the **Join** operator and its “**ON**” clause (if it exists).
2. **Comment** the **right** input record-list plus the parentheses and the “**AS**” clause of the **left** input record-list. **Run and debug** the **left** uncommented input record-list until it is **fixed**.
3. **Comment** the **left** input record-list plus the parentheses and the “**AS**” clause of the **right** input record-list. **Run and debug** the **right** uncommented input record-list until it is **fixed**.
4. **Uncomment** the **left** input record-list plus the parentheses and the “**AS**” clause of the **right** input record-list. You also uncomment the **Join** operation and the enclosing **Select** operation that you commented in step **1** above. **Run and debug** the now complete enclosing **Select** operation until it is fixed.

If you want more information about this, you may click:

- “*J.5 How do debug inside out?*”
- “*J.7 How do I debug the current uncommented SQL operation at each step?*”

## **J.7 How do I debug the current uncommented SQL operation at each step?**

At each debugging step you have a given uncommented **current uncommented SQL operation** (click *J.3*) that you want to debug. When you try to run it, you may encounter four different **types** of problems. I provide specific advice to **fix** each type of problem



of the **current uncommented SQL operation**, as follows:

1. You **cannot save** the Query. Click on:

- “*J.7.1 How do I debug the current uncommented SQL operation so it can be saved?*”
- “*J.8 How do I fix a syntax error that prevents saving a Query?*”

2. It saves well, but **crashes** when run. Click on:

- “*J.7.2 How do I debug the current uncommented SQL operation so it runs?*”
- “*J.9 How do I fix a crash from a syntax error?*”
- “*J.10 How do I fix a crash from a run-time error?*”

3. It saves and runs, but it produces **defective** results. Click on:

- “*J.7.3 How do I debug the current uncommented SQL operation so it does not produce defective results?*”
- “*J.11 How do I fix defective Query results?*”

4. It saves and runs well, but its **results** are **not** the ones that you want. Click on:

- “*J.7.4 How do I debug the current uncommented SQL operation’s functionality?*”

If you debug and debug, and just **cannot fix** the Query, I advise you do a workaround. You may click:

- “*J.12 What do I do when I just cannot fix a Query?*”.

### **J.7.1 How do I debug the current uncommented SQL operation so it can be saved?**

This section also answers the question:

- **Why cannot I save my Query?**

If MS-Access **does not allow you** to save the Query with the current uncommented SQL operation (click *J.3*) it is because:

- Almost always, the Query has **syntax** errors  
In this case, you will get a syntax error message providing some information about the syntax error.
- Very unusual, the Query is larger than **65,535** characters<sup>80</sup>  
In this case, you will get the error message “*Access SQL Editor is not able to process query ‘Query\_name’ because it is too long. Query size: nnnn characters, maximum size 65535 characters.*”

If you want to **debug** the error now, the **fix** depends on the error cause:

- If the cause is a **syntax** error, you **fix** it by checking:
  - “*J.8 How do I fix a syntax error that prevents saving a Query?*”.


---


<sup>80</sup> I am giving for granted that you are using the “**Access SQL Editor**”.

- If the cause is the Query is **larger than 65,535 characters**, you **must** make the Query smaller. You **fix** this by shortening some comments or moving part of the SQL code to an auxiliary Query invoked from this one or whatever other modification that makes the Query smaller than 65,535 characters.

If you rather want to **save** the Query **now** (i.e., you do not want to debug the error now), check the following two cases:

- If you are using the “**Access SQL Editor**” **and** the Query is **smaller than 65535** characters

You select **all** the SQL code in the Query by pressing “**Ctrl-a**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**a**” key) while the mouse pointer is over the corresponding “Query pane”. You then click on the **Comment** “ **Comment**” icon and all the SQL code will be commented (i.e., each line will be prefixed by two hyphens “--”). You then write at the first line of the Query code a simple “filler” **Select** operation that will not produce any syntax error. This will allow you to save the Query.

When you come back to debugging this Query, you open it, you delete the simple “filler” **Select** operation that you wrote, you select all the SQL code by pressing “**Ctrl-a**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**a**” key), and finally click on the **Uncomment** “ **Uncomment**” icon.

- If you are **not** using the “**Access SQL Editor**” **or** the Query is **larger than 65,535** characters

You **copy** the whole SQL Query code from the **Access SQL Editor** (or from the “Query pane” in “**SQL View**”), you paste it in some external plain text editor (e.g., notepad, textpad, emacs, vi, ...) and save it as a plain text file. Then you put in the Query code any correct SQL operation (e.g., a most simple **Select** operation) that will allow you to save the Query code for later debugging of its syntax errors. When you come back to debugging the Query, you copy the Query code from the text file and paste it into the “Query pane” in the “**Access SQL Editor**” (or into the “Query pane” in “**SQL View**”) and you retake the task of debugging it.

Be aware that **some types** of **syntax** errors prevent the Query from being saved, but **other types** of **syntax** errors allow to **save** the Query. Therefore, even if you could save the Query, it can **still contain syntax errors**. If you run a Query with **syntax errors** (or other severe errors) it will **crash**. If you want to **fix** this, you may click “*J.7.2 How do I debug the current uncommented SQL operation so it runs?*”.

Let me show how to **debug** a Query (i.e., how to **find** and **fix** the **errors**) with an example. Imagine that you have written the following Query<sup>81</sup> to show the amount of rainfall by city, by year and by quarter (where the auxiliary Query

---

<sup>81</sup> This is the Query “**J\_Debugging**” from file “**Company\_Database.accdb**”.

“K\_Rows\_into\_columns\_1” is the one from K.6.4):

```

SELECT Capital, Cal_Year, Val
      , Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4,, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (-- This is the inner select, producing four records
   SELECT Num, 5/(Num-1) AS Val FROM T_Numbrs WHERE Num BETWEEN 1 AND 4
  ) AS Quadr
'
K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

You try to **run** it, and MS-Access **does not allow you to save it**, while it shows the syntax error message “Syntax error, missing operator in query expression 'Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4,, "Q4")'.”.

You then **search** for the expression “Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4,, "Q4)” (from the error message), and you find it in line 2. You then comment that line of code (notice the **two green hyphens** at the beginning of line 2) and you get this **first “current uncommented SQL operation”**<sup>82</sup>:

```

SELECT Capital, Cal_Year, Val
      --, Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4,, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (-- This is the inner select, producing four records
   SELECT Num, 5/(Num-1) AS Val FROM T_Numbrs WHERE Num BETWEEN 1 AND 4
  ) AS Quadr
'
K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

Notice that the way the SQL code is formatted allows that when commenting line 2, the uncommented code fragment is a syntactically correct SQL operation (see the bullet point on parentheses from K.4.3). If the SQL code was not formatted in this way, you would have needed to do **some editing** in your code, **in addition to commenting**, to make sure that the **uncommented** code corresponds to a correct “**current uncommented SQL operation**”. Notice also that the **uncommented** lines are **not consecutive**, which is a quite frequent case.

The commented code above now **saves well**, so it confirms there is a syntax error in line 2. You carefully look into line 2, and you notice that there are two consecutive commas “,,” after “**Num=4**”. You remove the extra comma, and you also uncomment the second

---

<sup>82</sup> This is the Query “J\_Debugging\_1” from file “Company\_Database.accdb”.

line, and you get this *second* “**current uncommented SQL operation**”<sup>83</sup>:

```
SELECT Capital, Cal_Year, Val
      , Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (-- This is the inner select, producing four records
   SELECT Num, 5/(Num-1) AS Val FROM T_Numbrs WHERE Num BETWEEN 1 AND 4
   ) AS Quadr
'
K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")
```

The current uncommented SQL operation now **saves well!**

However, when you **run/execute** the current uncommented SQL operation, it **crashes** with the **syntax** error message:

*“The Microsoft Access database engine cannot find the input table or query 'T\_Numbrs'. Make sure it exists and that its name is spelled correctly.”*

As a side note, if you **run** the Query from the “**Access SQL Editor**” (click *F.5*) the error message will be different: “*You cancelled the previous operation.*”

I will continue debugging this same example at the end of the next section *J.7.2*.

### **J.7.2 How do I debug the current uncommented SQL operation so it runs?**

In order to **run**, the “**current uncommented SQL operation**” must be **syntactically correct** (i.e., it has to comply with the rules of writing SQL code) **and** it must **not** contain **coding errors** that make it **crash**.

In case the **current uncommented SQL operation** uses auxiliary Queries (i.e., the code of the **current uncommented SQL operation** contains **Query names**), you should **first** make reasonably sure that **all** these auxiliary Queries are correct. You know if the **current uncommented SQL operation** makes use of auxiliary Queries by visually inspecting **all** its “**FROM**” clauses or, most reliably, by using an object dependency checker (click *I.3*). My advice is that you **trust** the **test-and-proven** auxiliary Queries (i.e., the ones that have been working well for months), and **do not** trust the **non-test-and-proven** auxiliary Queries. You should therefore **first** debug all the **non-test-and-proven** auxiliary Queries used in the **current uncommented SQL operation** before starting to debug the code of the **current uncommented SQL operation** itself.

Once you have finished debugging the auxiliary Queries that are **non-test-and-proven**, you can begin debugging the crash in the **current uncommented SQL operation**.

A crash may arise from a **syntax** error (i.e., from the writing rules of SQL) or from a **run-time** error (i.e., from the processing of the database values).

Regardless of a **crash** arising from a **syntax** error or from a **run-time** error, you will anyway have to locate and fix the error. However, it is **useful** to know the difference because **syntax** errors are found and fixed in a different way than **run-time** errors. Main

---

<sup>83</sup> This is the Query “**J\_Debugging\_2**” from file “**Company\_Database.accdb**”.

differences are the following:

- **Syntax** errors always manifest **before run-time** errors. In order to get a **run-time** error, the current uncommented SQL operation **must** already be syntactically correct.
- **Syntax** error messages frequently provide more information about the piece of code that originated them, so you can **search** in the current uncommented SQL operation for the corresponding expression of code fragment. They are therefore easier to locate than **run-time** errors.
- **Syntax** errors arise from not having followed the SQL writing rules (i.e., wrong syntax), so you already have some idea of what their cause is. However, **run-time** errors may arise from very complex interrelations between data and SQL-code and can manifest themselves in segments of code that are away from their actual cause.

In order to **fix** the crash, you may look for the **crash error message** that you got in one of the two following sections, depending on whether it is a **syntax** error message or a **run-time** error message:

- “*J.9 How do I fix a crash from a syntax error?*”
- “*J.10 How do I fix a crash from a run-time error?*”

If you debug and debug, and just cannot fix the Query, I advise you do a workaround. You may click:

- “*J.12 What do I do when I just cannot fix a Query?*”.

Let me show you this continuing with the example from the previous section *J.7.1*. Remind that the Query **saved well**, but when you **run/execute** it, it **crashes** with the **syntax** error message:

*“The Microsoft Access database engine cannot find the input table or query 'T\_Numbrs'. Make sure it exists and that its name is spelled correctly.”*

You notice that the current uncommented SQL operation uses the auxiliary Query “**K\_Rows\_into\_columns\_1**”. Let us suppose this is a **test-and-proven** Query, so you do not debug it, and proceed instead to debug the current uncommented SQL operation.

You search for “*T\_Numbrs*” (from the error message) and you find it in line 6. You now comment lines 5 to 8 and 10 to 11, because if you only comment line 6, you would **not** get a correct **current uncommented SQL operation**. Commenting lines 5 to 8 and 10 to 11 you get this **third current uncommented SQL operation**<sup>84</sup>:

```

SELECT Capital, Cal_Year, Val
        , Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
        , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
    --(-- This is the inner select, producing four records
    --SELECT Num, 5/(Num-1) AS Val FROM T_Numbrs WHERE Num BETWEEN 1 AND 4
    --) AS Quadr
    --,
    K_Rows_into_columns_1
--ORDER BY Capital, Cal_Year
        --, Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

---

<sup>84</sup> This is the Query “**J\_Debugging\_3**” from file “**Company\_Database.accdb**”.

This now **runs** (although it requests parameters “Val” and “Num”), so it confirms there is a syntax error in line 6, in Table name “**T\_Numbrs**”. You carefully look at it, and you notice it is misspelled. You replace it by “**T\_Numbers**” and also uncomment lines 5 to 8 and 10 to 11, getting this **fourth current uncommented SQL operation**<sup>85</sup>:

```

SELECT Capital, Cal_Year, Val
      , Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  ( -- This is the inner select, producing four records
    SELECT Num, 5/(Num-1) AS Val FROM T_Numbers WHERE Num BETWEEN 1 AND 4
    ) AS Quadr
  ,
  K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

This **current uncommented SQL operation** now **runs!!**

However, its results show the divide-by-zero “**#Div/0!**” exception-value, so you are having **defective** results. I will continue debugging this same example at the end of the next section.

The Table “T\_Numbers” used in the example above is an auxiliary Table that just contains integer numbers (click *K.2.2*).

### **J.7.3 How do I debug the current uncommented SQL operation so it does not produce defective results?**

The **current uncommented SQL operation** produces **defective** results. When I say that results are **defective** it is not that they are **different** from what you intended, but rather that they are actually **flawed**. Some examples of **defective** results are:

- Showing **exception-values**.
- Requesting a parameter you **did not** define.
- Requesting the **same** parameter **twice**.
- Considering **two different** values as equal (e.g., in a “**WHERE**” *Boolean* expression).
- **Wrong** record **sorting**.
- **Arithmetic errors**.

Regardless of the type of defective result, you must debug it/them from your **current uncommented SQL operation**.

If you want to **fix** these defective results, you may click:

- “*J.11 How do I fix defective Query results?*”

If you debug and debug, and just cannot fix the Query, I advise you do a workaround checking:

- “*J.12 What do I do when I just cannot fix a Query?*”

Let me show you this continuing with the example from the previous section *J.7.2*. Remind that the current uncommented SQL operation **saved well**, but when you

---

<sup>85</sup> This is the Query “**J\_Debugging\_4**” from file “**Company\_Database.accdb**”.

**run/execute** it, it shows the divide-by-zero “#Div/0!” exception-value, so you are having **defective** results.

You search for a division operator “/” and you find it in line 6, that contains the expression “5/(Num-1)”. This expression causes division by zero when “Num=1”. Replacing “Num-1” by “Num”, you get this *fifth* current uncommented SQL Operation<sup>86</sup>:

```
SELECT Capital, Cal_Year, Val
      , Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (
    (-- This is the inner select, producing four records
     SELECT Num, 5/Num AS Val FROM T_Numbers WHERE Num BETWEEN 1 AND 4
    ) AS Quadr
  )
  K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")
```

This now **saves and runs well!!**

However, this current uncommented SQL operation **does not** produce the amount of rainfall by city, by year and by quarter, which is what you wanted. I will continue debugging this same example at the end of the next section *J.7.4*.

The Table “T\_Numbers” used in the example above is an auxiliary Table that just contains integer numbers (click *K.2.2*).

### **J.7.4 How do I debug the current uncommented SQL operation’s functionality?**

What happened in the example along the previous three sections is not something I forced, and rather it is the **most usual** situation: you solve a few crashes but once the code runs, you have to debug its functionality until it actually **produces the results you want**. Therefore, once you get an SQL operation to run without defective results, you are **not done** debugging, and you will most likely have to do **substantially more** debugging.

The procedure to debug the **functionality** is basically the same as the one for debugging syntax and run-time errors that I showed in the previous sections: you have to progressively comment/uncomment your SQL code, correcting the **functionality**, until you debug your current uncommented SQL operation, and it produces the results that you want.

If you debug and debug, and just cannot fix the Query, I advise you do a workaround checking:

- “*J.12 What do I do when I just cannot fix a Query?*”

Let me show how to fix the **functionality** of an SQL operation continuing with the example from the previous section, where we finished with the following *fifth* current

---

<sup>86</sup> This is the Query “J\_Debugging\_5” from file “Company\_Database.accdb”.



uncommented SQL operation<sup>87</sup>:

```

SELECT Capital, Cal_Year, Val
      , Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (-- This is the inner select, producing four records
   SELECT Num, 5/Num AS Val FROM T_Numbers WHERE Num BETWEEN 1 AND 4
   ) AS Quadr
,
  K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

This Query should produce the amount of rainfall by city, by year and by quarter, but it currently produces **no Q2 results**, and **Q1 results are duplicated**. The way I advise you debug this is the following:

You **run** the auxiliary Query “**K\_Rows\_into\_columns\_1**” and check (just in case) its results. They are correct, so the error is not there.

You now comment the lines 1 to 5 and 7 to 11, getting the *sixth* current uncommented SQL Operation<sup>88</sup>:

```

--SELECT Capital, Cal_Year, Val
      --, Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
      --, Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
--FROM
  --(-- This is the inner select, producing four records
   SELECT Num, 5/Num AS Val FROM T_Numbers WHERE Num BETWEEN 1 AND 4
   --) AS Quadr
--,
  --K_Rows_into_columns_1
--ORDER BY Capital, Cal_Year
      --, Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

You check the expected results (the numbers 1 to 4), and you verify that they are correct. Since the **two input** record-lists to the **Cross-Join** are correct, you then conclude that the error is in the outermost “**SELECT**” clause. You decide to comment part of the “**SELECT**” **expressions** and check their results: you uncomment all the lines, and comment only line 2, getting the *seventh* current uncommented SQL Operation<sup>89</sup>:

```

SELECT Capital, Cal_Year, Val
      --, Switch(Num=1,"Q1", Num=2,"Q1", Num=3,"Q3", Num=4, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (-- This is the inner select, producing four records
   SELECT Num, 5/Num AS Val FROM T_Numbers WHERE Num BETWEEN 1 AND 4
   ) AS Quadr
,
  K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

Running this code you can check that the actual **values** for the rainfall by city, by year and by quarter are correct. Therefore, the error must be in the **names** of the Quarters, which are in line 2, and are the only ones not checked. You carefully look at them, you notice that in line 2 the second “**Q1**” should be a “**Q2**”, and you fix it. The resulting

<sup>87</sup> This is the Query “**J\_Debugging\_5**” from file “**Company\_Database.accdb**”.

<sup>88</sup> This is the Query “**J\_Debugging\_6**” from file “**Company\_Database.accdb**”.

<sup>89</sup> This is the Query “**J\_Debugging\_7**” from file “**Company\_Database.accdb**”.



*eighth* current uncommented SQL Operation<sup>90</sup> is:

```

SELECT Capital, Cal_Year, Val
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4") AS Quart
      , Switch(Num=1,Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (-- This is the inner select, producing four records
   SELECT Num, 5/Num AS Val FROM T_Numbers WHERE Num BETWEEN 1 AND 4
   ) AS Quadr
'
K_Rows_into_columns_1
ORDER BY Capital, Cal_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4, "Q4")

```

which finally **saves well, runs well** and has the **desired functionality!!**

The Table “T\_Numbers” used in the examples above is an auxiliary Table that just contains integer numbers (click *K.2.2*).

### **J.7.5 How do I check the results of the current uncommented SQL operation?**

At each debugging step you need to check if the results of the fragment SQL operation are correct. SQL operations return a record-list which may contain a large amount of data. Going over all the data with **visual inspection** to check it if is correct is feasible on some cases. However, if the SQL operation results are too large and complex, checking their correctness with visual inspection is slow, cumbersome and error prone. I now present several **complementary** approaches that can help you check the correct functionality of each SQL operation:

- **Debug the Queries in a duplicate database**  
Produce a copy of the database Tables and database Queries (in a split database this implies copying the **frontend files** and **source files** and **relinking** the linked Tables, click *K.3.10*) and use it to develop/debug your new Queries. In this way you can modify the data in the Tables as you need (deleting records, adding records useful for tests, ...) to produce the results that are most convenient for the debugging process.
- **Restrict the input record-lists of your Query**  
In the innermost **Select** operations of your Query, the ones that are taking the input-record-lists for the Query, replace their “**WHERE**” clauses by others that restrict the input records to the ones that are useful for your debugging. When you are done debugging, you replace the testing “**WHERE**” clauses by the original “**WHERE**” clauses. The usual way to do this is **commenting** each original “**WHERE**” clause, adding a new **testing** “**WHERE**” clause, and adding **very clear comments** to the SQL code, so you can undo this without making mistakes. This a good technique but you have to be **very careful** and **always mark with searchable comments** all the original (commented) “**WHERE**” clauses and all the testing “**WHERE**” clauses. By “searchable” comments I mean that you put in all these comments some codeword (e.g., “**REMOVE\_ME**”) to clearly identify these pieces of code. Doing this will allow you to search for the specific codeword you used in your comments to mark these pieces of code and undo the changes without mistakenly missing any of them.

---

<sup>90</sup> This is the Query “**J\_Debugging\_8**” from file “**Company\_Database.accdb**”.

- **Check the results with a record-list comparator** (click *J.13.1*)  
Whenever you have a record-list of your expected results (which is sometimes the case), use a **record-list comparator** tool to check the **actual** record-list produced by the SQL operation against the **expected** record-list that you have. The **record-list comparator** tool will automatically **identify and mark** all the possible differences, greatly simplifying your work, and greatly reducing the possibility of you overlooking some of the differences.

## **J.8 How do I fix a syntax error that prevents saving a Query?**

You can look for the syntax error message you got in the following list of **syntax error messages** (in alphabetical order). Once you find it, you may check its corresponding causes and fixes:

- *“Extra ) in query expression 'expression()'.”*  
This is most likely because there is an extra closing parenthesis “)” in the expression “*expression()*”, that is listed explicitly in the error message.  
You **fix** this by **searching** for “*expression()*” (from the error message) in your Query code and correcting it.
- *“Invalid SQL statement; expected 'DELETE', 'INSERT', 'PROCEDURE', 'SELECT', or 'UPDATE'.”*  
You **fix** this by correcting one of the following possible errors:
  - A mistyped or missing “**SELECT**” keyword.
  - A mistyped or missing keyword among the other ones listed in the error message.
- *“JOIN expression not supported.”*  
This is most likely caused by a wrong **Boolean** expression in the “**ON**” clause (click *F.8.9*) of the **Join operation**. You **fix** this by correcting one of the following possible errors:
  - One or more field names in the “**ON**” **Boolean** expression are not qualified with the **name** of the corresponding input record-list of the **Join operation**.
  - One (or more) **Boolean** sub-expressions **within** the “**ON**” **Boolean**-expression does not contain field names from **both** input record-lists of the **Join operation**.
- *“Syntax error, missing operator in query expression 'expression()'.”*  
You **fix** this by **searching** for “*expression()*” (from the error message) in your Query code and correcting one of the following possible errors:
  - Extra opening parenthesis “(”.
  - Wrong parentheses matching.
  - Missing comma “,” between two arguments of a function.
  - You are using both “**DISTINCT**” and “**DISTINCTROW**”.
  - Missing operator.
- *“Syntax error in FROM clause.”*  
You **fix** this by **locating** the wrong “**FROM**” clause, and correcting one of the following possible errors:
  - You enclosed a Table name or Query name between parentheses and also used the “**AS**” clause: either remove the parentheses or the “**AS**” clause.

- You did not enclose between parentheses a **Select** operation: enclose it between parentheses.
  - Wrong parentheses matching at the end of a **Select** operation.
  - Extra commas between fields.
  - Wrong number of arguments in functions.
  - Not having enclosed in parentheses each of the two SQL operations that are the operands of a **Join** operator.
  - Misspelled operator (e.g., “**MOOD**” instead of “**MOD**”).
  - Error in “**ON**” expression.
  - You used a comma “,” instead of a period “.” to separate decimals in numbers. This error is particularly frequent if in your country you use comma “,” to separate decimals.
  - You are doing a “**SELECT**” over a “**TRANSFORM**”.
- “*Syntax error in TRANSFORM clause.*”  
You **fix** this by removing the “**HAVING**” clause from the Transform operation. Notice that there may be other possible errors.
  - “*The SELECT statement includes a reserved word or an argument name that is misspelled or missing, or the punctuation is incorrect.*”  
You **fix** this by correcting one of the following possible errors:
    - You are using a reserved keyword (e.g., “**Value**”) as a field name or SQL operation name.
    - You used an invalid parameter in the “**TOP**” clause. Invalid parameters are expressions (i.e., not a constant), zero or negative constants, and for the “**TOP PERCENT**” also constants greater than 100.

If you want detailed guidance on the **writing rules** (syntax) of SQL operations, you may click:

- “*F.7.15 How do I write a correct (syntax) Select-no aggreg?*”
- “*F.7.16 How do I write a correct (syntax) Select-group by aggreg?*”
- “*F.7.17 How do I write a correct (syntax) Select-total aggreg?*”
- “*F.8.10 How do I write a correct (syntax) Join?*”
- “*F.9.5 How do I write a correct (syntax) Union?*”
- “*F.10.14 How do I write a correct (syntax) Transform?*”
- “*F.12 How do I add parameters (type-in variables) to my Queries?*”
- “*F.13 How do I write a Query that changes my Table data?*”

If you want guidance on writing SQL operations, you may click:

- “*K.4 What Query design principles should I follow?*”.

## **J.9 How do I fix a crash from a syntax error?**

Some Query **crashes** are due to **syntax** errors: the Query is **not properly written**, and it **crashes** before it is actually being run (it crashes in the parsing phase of the SQL code).

All **crash** error messages that begin with “*Syntax error...*” are caused by a **syntax** error (that was easy!). **Crash** error messages that **do not** begin with “*Syntax error...*”, but whose **content** refers to errors in the rules for writing SQL code are also caused by a **syntax** error. Some examples of such **crash syntax** error messages are:

- “*Cannot repeat table name 'Table\_name'.*”
- “*The specified field 'Field\_name' could refer to more than one table listed in the FROM clause of your SQL statement.*”
- “*Wrong data type for parameter '[Param\_name]'.*”

You can look for the **crash** error message you got in the following list of **syntax error messages** (in alphabetical order). Once you find it, you may check its corresponding causes and fixes:

- “*Cannot group on fields selected with '\*'.*”  
The most likely cause is a “**SELECT \***” with either a “**GROUP BY**” clause. You **fix** this by searching for “**SELECT \***” to locate the wrong **Select**, and then writing an **explicit** list of “**SELECT**” **expressions** (and typically also **output field names**) after the “**SELECT**” keyword.  
If you want to know more, you may click “[K.4.9 Why should I avoid using “SELECT \\*”?](#)”.
- “*Cannot have aggregate function in expression 'exp()'.*”  
The most likely cause is nested SQL aggregate functions. You **fix** this by **searching** for “*exp()*” (from the error message) to locate the wrong expression, and then **removing** the SQL aggregate function you are **erroneously** using **inside** another SQL aggregate function.  
You can **never** use an SQL aggregate function inside the argument expression of **another** SQL aggregate function.
- “*Cannot have aggregate function in WHERE clause 'exp()'.*”  
The most likely cause is an SQL aggregate function inside a “**WHERE**” expression. You **fix** this by **searching** for “*exp()*” (from the error message) to locate the wrong “**WHERE**” expression, and then **removing** the SQL aggregate function you are **erroneously** using **inside** it. You can **never** use an SQL aggregate function in a “**WHERE**” expression.
- “*Cannot repeat table name '%\$##@\_Alias' in FROM clause.*”  
The most likely cause is a **Join** operation where **both** of its input SQL operations have **not** been assigned a name using the “**AS**” clause. For this reason, MS-Access assigned the name “*\$\$##@\_Alias*” to **both** of them, and this is why the name is **repeated**.  
You **fix** this by **finding** the wrong **Join** operation and **assigning a name to at least one** (better to both) of its two input SQL operations. You assign a name to an input SQL operation by enclosing it between parentheses, followed by the “**AS**” keyword and followed by a **name** (the assigned **name must be different** in both SQL operations).  
If you want to know more, you may click “[F.8.10.2 What are the formal rules \(syntax\) to write a Join?](#)”.

- “Cannot repeat table name 'Table\_name'”  
The most likely cause is a **Join** operation with the **same name** “Table\_name” assigned to **both** of its input record-lists.  
You **fix** this by **searching** for “Table\_name” (from the error message) to locate the wrong **Join** operation, and then assigning **different names** to its **two** input record-lists.  
If you want to know more, you may click “[F.8.10.2 What are the formal rules \(syntax\) to write a Join?](#)”.
- “Circular reference caused by alias 'Alias\_name' in query selection's SELECT list.”  
The most likely cause is a **circular** field name reference in “**SELECT**” **expressions**.  
You **fix** this by **searching** for “Alias\_name” (from the error message) to locate the wrong “**SELECT**” **expressions** where you are assigning an output field name to **itself**, or you are assigning output field names in a circular way (e.g., “Name\_1” is used in the expression for “Name\_2”, “Name\_2” is used in the expression for “Name\_3” and “Name\_3” is used in the expression for “Name\_1”). Once you find this, you **change** the naming to suppress the circular reference.  
If you want to know more, you may click “[F.7.14 How do I write a correct \(syntax\) Select?](#)”.
- “Column names in each table must be unique. Column name 'Col\_name' is specified more than once.”  
The most likely cause is a **duplicate value** in the “**IN**” **list-of-PIVOT-values** of the **Transform** operation.  
You **fix** this by **making sure** that the “**IN**” **list-of-PIVOT-values** **does not contain any duplicate** value.  
If you want to know more, you may click “[F.10.12 What is the “IN” clause of a Transform?](#)”.
- “HAVING clause (exp()) without grouping or aggregation.”  
The most likely cause is a “**HAVING**” clause in a **Select-no\_agg** (this is, a **Select** without a “**GROUP BY**” clause and without SQL aggregate functions).  
You **fix** this by **searching** for “exp()” (from the error message) to locate the wrong **Select-no\_agg** operation. If you **do not** want aggregation in this **Select** operation, you then remove the “**HAVING**” clause by either turning it into a “**WHERE**” clause, or by integrating the “**HAVING**” **Boolean** expression into the “**WHERE**” **Boolean** expression. If you **do want** aggregation in this **Select** operation, you then add the corresponding “**GROUP BY**” clause and/or SQL aggregate functions.  
If you want to know more, you may click “[F.7.16.2 What are the formal rules \(syntax\) to write a Select-group by agg?](#)”.
- “Item not found in this collection.”  
Some possible causes are the following:
  - Not having enclosed between parentheses each of the two SQL operations that are the operands of a **Join** operator.
  - Problems with identifiers.

- “Number of query values and destination fields are not the same.”  
The most likely cause is an **Insert-one-record** operation where either:
  - The **number** of “**VALUES**” expressions is **different** from the **number** of **fields** in the **list of target Table field names**.  
You **fix** this by making **both** of them having **the same number** of elements.
  - The **number** of “**VALUES**” expressions is **different** from the **number** of **target Table field names** (and the optional **list** is **not** used).  
You **fix** this by making the **number** of “**VALUES**” expressions the same as the **number** of **target Table field names**.

In either case you fix this by making **both** of them having **the same number** of elements. If you want to know more, you may click “[F.13.2.2 How do I write a Query that inserts only one Table record?](#)”.

- “**ORDER BY** clause (exp()) conflicts with **DISTINCT**.”  
The most likely cause is a **Select** with “**DISTINCT**” where one (or more) of the “**ORDER BY**” expressions is/are not **exactly the same** as one of the “**SELECT**” expressions.  
You **fix** this by **searching** for “exp()” (from the error message) to locate the wrong **Select** operation, then you **remove** its “**ORDER BY**” clause, and finally you **enclose** this **Select** into a **new Select** operation where you can write the “**ORDER BY**” expressions that you want.  
If you want to know more, you may click “[F.7.12 How do I use “ORDER BY” to order the output records of a Select?](#)”.
- “**ORDER BY** clause (exp()) conflicts with **GROUP BY**.”  
The most likely cause is a **Transform** where one (or more) of the “**ORDER BY**” expressions is/are not **exactly the same** as either of the “**GROUP BY**” expressions or as the “**PIVOT**” expression.  
You **fix** this by making (in the **Transform**) **each and every** of the “**ORDER BY**” expressions to be **exactly the same** as either of the “**GROUP BY**” expressions or as the “**PIVOT**” expression.  
If you want to know more, you may click “[F.10.8 What is the “ORDER BY” clause of a Transform?](#)”.
- “Subqueries cannot be used in the expression 'Agg\_func()'”  
The most likely cause is nested SQL aggregate functions.  
You **fix** this by **searching** for “Agg\_func()” (from the error message) to locate the wrong SQL aggregate function, and **removing** the SQL aggregate function you are using **inside** its argument.  
You can **never** use an SQL aggregate function inside the argument of another SQL aggregate function.
- “The **INSERT INTO** statement contains the following unknown field name: 'Field\_name'. Make sure you have typed the name correctly, and try the operation again.”  
The most likely cause is a **mistyped output** field name in the “**SELECT**” clause of the **Insert-many-records** operation.  
You **fix** this by correcting the wrong **output** field name(s) from the “**SELECT**” clause so that **each and every** of them corresponds to one **target Table field name**.

If you want to know more, you may click “[F.13.2.1 How do I write a Query that inserts many Table records?](#)”.

- “The Microsoft Access database engine cannot find the input table or query 'T\_Q\_name'. Make sure it exists and that its name is spelled correctly.”

The most likely cause is a **mistyped** or **outdated** Table/Query name.

You **fix** this by searching for “T\_Q\_name” (from the error message), and then correcting its name.

If this is not the case, then it is not an actual syntax error, but I will anyway explain here how to fix it.

If it is a **linked** Table, and its name was correctly written in the Query code, the most likely cause is a failed Table **link**.

You **fix** this by correcting the Table **link**: see “[K.3.10 How do I view and manage Table links?](#)”.

In other case, the most likely cause is that a Table/Query was deleted by mistake, which is a **very severe** problem.

You **fix** this by recovering the deleted Table/Query from your **backup** files.

- “The number of columns in the two selected tables or queries of a union query do not match.”

The most likely cause is a **Union** operation with a **different number of fields** (i.e., different number of columns) in its two input record-lists.

You **fix** this by finding the wrong **Union** operation and making both its input record-list having the same number of fields.

- “The specified field 'Field\_name' could refer to more than one table listed in the FROM clause of your SQL statement.”

The most likely cause is that you **did not qualify** (click C.2.2) a field name used in a “**SELECT**” **expression** and that field name exists in **both** input record-lists of the enclosed **Join** operation.

You **fix** this by **searching** for “Field\_name” (from the error message) to locate the wrong “**SELECT**” **expression**, and then you correctly **qualify** that field name.

If you want to know more, you may click “[F.7.5 What are the output fields \(“SELECT” clause\) of a Select?](#)”.

- “Undefined function 'WrongFunc' in expression.”

The most likely cause is a **mistyped** function name.

You **fix** this by **searching** for “WrongFunc” (from the error message) to locate the wrong function name, and then you correct it.

- “Wrong data type for parameter '[Param\_name].”

The most likely cause is two parameters with the **same name** but with a **different data type**.

You **fix** this by searching for “Param\_name” (from the error message) in the current Query, and in **all** its auxiliary Queries (**recursively**) to locate the colliding parameters, and then you change the name of one of them.

This may happen when you declare a parameter in a Query, and the parameter **name** that you selected is **the same** as the one of another parameter declared inside an auxiliary Query, and both parameters have a **different** data type.

Notice that if both parameters have the **same data type** there is no **crash**, because they are considered as the **same parameter**.



If you want to know more, you may click *“F.12 How do I add parameters (type-in variables) to my Queries?”*.

- *“Wrong number of arguments with function in query expression 'Wrong\_exp()'.”*  
The most likely cause is a wrong number of arguments in one (or more) of the functions inside the expression *“Wrong\_exp()”*.  
You **fix** this by **searching** for the expression *“Wrong\_exp()”* (from the error message) to locate the wrong function invocation(s), and then using the correct number of arguments.
- *“You cancelled the previous operation.”*  
This is a **misleading** error message shown by the **“Access SQL Editor”** (click *F.5*). If you get this error message, my advice is you run the Query from MS-Access, and you will then get the error message *“The Microsoft Access database engine cannot find the input table or query 'T\_Q\_name'. Make sure it exists and that its name is spelled correctly.”*. You will find this error message higher above in this list with an explanation on how to **fix** the problem.
- *“Your Query does not include the specified expression 'exp()' as part of an aggregate function.”*  
The most likely cause is that you are using **wrong elements** to build a **“TRANSFORM”**, **“SELECT”**, **“HAVING”** and/or **“ORDER BY”** expression(s) belonging to a **Select-group\_by\_aggreg** operation or a **Transform** operation.  
You **fix** this by **searching** for the expression *“exp()”* (from the error message) to **locate** the wrong **Select-group\_by\_aggreg** operation or wrong **Transform** operation. The expression *“exp()”* can be inside their **“TRANSFORM”**, **“SELECT”**, **“HAVING”** and/or **“ORDER BY”** expression(s). Once you have **located** the wrong **Select-group\_by\_aggreg** or **Transform** operation, you **correct** its **“TRANSFORM”**, **“SELECT”**, **“HAVING”** and/or **“ORDER BY”** expression(s) by using **only** the corresponding **allowed elements**. For a **Select-group\_by\_aggreg** operation you may click *“F.7.16 How do I write a correct (syntax) Select-group by aggreg?”*. For a **Transform** operation you may click *“F.10.14 How do I write a correct (syntax) Transform?”*.

If the crash error message did **not** begin with *“Syntax error...”* and is **not** in the list above, then the crash is most likely from a **run-time error**. You may therefore **fix** it clicking *“J.10 How do I fix a crash from a run-time error?”*.

If you want detailed guidance on the **writing rules** (syntax) of SQL operations, you may click:

- *“F.7.15 How do I write a correct (syntax) Select-no aggreg?”*
- *“F.7.16 How do I write a correct (syntax) Select-group by aggreg?”*
- *“F.7.17 How do I write a correct (syntax) Select-total aggreg?”*
- *“F.8.10 How do I write a correct (syntax) Join?”*
- *“F.9.5 How do I write a correct (syntax) Union?”*
- *“F.10.14 How do I write a correct (syntax) Transform?”*
- *“F.12 How do I add parameters (type-in variables) to my Queries?”*



- “F.13 How do I write a Query that changes my Table data?”

If you want guidance on writing SQL operations, you may click:

- “K.4 What Query design principles should I follow?”.

## **J.10 How do I fix a crash from a run-time error?**

You can look for the **crash** error message you got in the title of the following sections. Once you find it, you may click on the corresponding section title to get advice on possible causes and fixes for the **crash**:

- “J.10.1 How do I fix the crash “Divide by zero.”?”
- “J.10.2 How do I fix the crash “Overflow.”?”
- “J.10.3 How do I fix the crash “Data type mismatch in criteria expression.”?”
- “J.10.4 How do I fix the crash “Data type mismatch.”?”
- “J.10.5 How do I fix the crash “Invalid Procedure Call or Argument.”?”
- “J.10.6 How do I fix the crash “Invalid use of Null.”?”
- “J.10.7 How do I fix the crash “The expression cannot be used in a calculated column.”?”
- “J.10.8 How do I fix the crash “Cannot have ... in aggregate argument.”?”
- “J.10.9 How do I fix the crash “Could not find field...”?”
- “J.10.10 How do I fix the crash “The expression cannot be used in a Calculated column”?”
- “J.10.11 How do I fix the crash “Expression is too complex.”?”
- “J.10.12 How do I fix the crash “Query is too complex.”?”
- “J.10.13 How do I fix the crash “Cannot open any more databases.”?”
- “J.10.14 How do I fix a crash from my user-defined VBA functions?”

### **J.10.1 How do I fix the crash “Divide by zero.”?**

If this is a **test-and-proven Query**, the **most likely** cause for the crash is that **Table data is inconsistent**. You **fix** this by clicking “J.1 How do I fix an error/crash in a test-and-proven Query?”.

If this is a **non-test-and-proven Query**, the crash is most likely because of one (or more) of the following:

- A divide-by-zero “**#Div/0!**” exception-value is an argument to a **function** (e.g., “Log (2/0)”).
- A divide-by-zero “**#Div/0!**” exception-value is an argument to the “**IN**” operator.
- A divide-by-zero “**#Div/0!**” exception-value is an an argument to a **domain** or **SQL aggregate function**.
- A **Union operation** where one (or both) of its SQL operations produces at least one

divide-by-zero “#Div/0!” exception-value in its record-list.

- A **user-defined** VBA function that contained a division by zero.

You **fix** these problems by preventing divide-by-zero “#Div/0!” in your Query: click “*J.11.10 How do I fix a Query producing “#Num!”, “#Div/0!”, “#Error”, “#Type!” or “#Func!” in a field?*”.

### J.10.2 How do I **fix** the **crash** “**Overflow.**”?

If this is a **test-and-proven Query**, the **most likely** cause for the crash is that **Table data is inconsistent**. You **fix** this by clicking “*J.1 How do I fix an error/crash in a test-and-proven Query?*”.

If this is a **non-test-and-proven Query**, the crash is most likely because of one (or more) of the following:

- A number-overflow “#Num!” exception-value is an argument to a **function** (e.g., “Log(0/0)”).
- A number-overflow “#Num!” exception-value is an argument to the “**IN**” operator.
- A number-overflow “#Num!” exception-value is an argument to a **domain** or **SQL aggregate function**.
- You are using **correct** values as arguments to a **numeric-like aggregate function**, but the correct values cause the aggregate function **itself** to overflow. Remind that the **numeric-like** aggregate functions are the SQL aggregate functions “**Sum()**”, “**Avg()**”, “**StDev()**”, “**StDevP()**”, “**Var()**” and “**VarP()**”, plus their corresponding domain aggregate functions “**DSum()**”, “**DAvg()**”, “**DStDev()**”, “**DStDevP()**”, “**DVar()**” and “**DVarP()**”.
- A **Union operation** where one (or both) of its SQL operations produces at least one number-overflow “#Num!” exception-value in its record-list.
- A **type-conversion function** over an argument that is out of the range of the target data/field type. For example, “**CInt(36000)**”, “**CByte(256)**” or “**CInt(-32769)**”.
- An expression caused overflow. Notice that **some** expressions that cause overflow crash the Query, while **others** do not crash the Query and their result will be shown as number-overflow “#Num!”. An example of an expression that **will crash** the Query is “1.7E308 + CDb1(1.7E308)”. An example of an expression that **will not crash** the Query is “1.7E308 + 1.7E308”. Overflow of operators over the **Currency** data/field type typically crash the Query.
- **All arithmetic operators** that perform **overflow** with one “**Ccur()**” type conversion function as one operand, and an **integer-like** value as the other operand, will cause a Query **crash**. For example, the expression:  

$$\text{CCur}(922337203685477.5807) + 1$$
will **crash** the Query, because the constant “**1**” is interpreted as **Long** (because it is combined with **Currency** and it is within the range of **Long**). You could expect this expression to return the exception-value number-overflow, but it does not, and instead it crashes the Query. As another example, the expression:  

$$\text{CCur}(922337203685477.5807) + 2147483648$$
**will not crash** the Query, because the constant “2147483648” is interpreted as

*Variant-Decimal* (because it is outside of the range of *Long*), and the expression will return the correct value “922339351169125.5807”.

- All arithmetic operators that perform overflow with one “CDBL()” type conversion function as one operand, will cause a Query crash showing the error message “Overflow.”. For example, the expression “Cdbl(1.7E308) + 1.7E308” will crash the Query.
- A user-defined VBA function that contained an operation producing overflow (either a variable assignment or a function argument out of range, or an expression causing overflow).

You fix these problems by preventing number-overflow “#Num!” in your Query: click “J.11.10 How do I fix a Query producing “#Num!”, “#Div/O!”, “#Error”, “#Type!” or “#Func!” in a field?”. Other problems you fix by correcting the corresponding expression.

### J.10.3 How do I fix the crash “Data type mismatch in criteria expression.”?

If this is a **test-and-proven Query**, the **most likely** cause for the crash is that **Table data is inconsistent**. You fix this by clicking “J.1 How do I fix an error/crash in a test-and-proven Query?”.

If this is a **non-test-and-proven Query**, the crash is most likely because of one (or more) of the following:

- A **Null** has been used as an argument to a **function**, where the function argument data type is other than *Variant* (e.g., “Log(Null)”).
- A **numeric-like aggregate function** is used over **Short Text, String, Long Text** (formerly “Memo”) or **Attachment** data/field types. Remind that the **numeric-like** aggregate functions are the SQL aggregate functions “Sum()”, “Avg()”, “StDev()”, “StDevP()”, “Var()” and “VarP()”, plus their dual domain aggregate functions “DSum()”, “DAvg()”, “DStDev()”, “DStDevP()”, “DVar()” and “DVarP()”.
- A **function invocation** with a data type mismatch between the provided value(s) and their corresponding function argument(s). For example, “Log(“Text”)”.
- An “IN” operator with a data type mismatch. Some examples of this are **Boolean** searched in a list of **String** and **Boolean**; **Integer** searched in a list of **String** and **Integer**; **String** searched in a list of **Integer** and **String**; **String** searched in a list of **Fractional** and **String**. For example:
  - **Boolean** searched in a list of **String** and **Boolean**:  
True IN (“Text\_string”, false)
  - **Integer-like** searched in a list of **String** and **integer-like**:  
12345 IN (“Text\_string”, 123)
  - **String** searched in list of **integer-like** and **String**:  
“Text\_string” IN (123, “Text\_string”)
  - **String** searched in list of **fractional** data type and **String**:  
“Text\_string” IN (12.34, “Text\_string”)
- A type-error “#Error” exception-value is an argument to a **function** (e.g.,

“Log ("Text"+1)”).

- A type-error “#Error” exception-value is an argument to the “IN” operator.
- A type-error “#Error” exception-value is an argument to a **domain** or **SQL aggregate function**<sup>91</sup> (e.g., “Sum ("Text"+1)”).
- A name-error “#Name?” exception-value is an argument to an **SQL aggregate function** other than “Count (\*)”, “Count ()”, “First ()”, and “Last ()”.
- A **Union operation** where one (or both) of its SQL operations produces at least one type-error “#Error” exception-value in its record-list.
- A **Null in some Union operations**. For example, the following Query<sup>92</sup> crashes:

```
SELECT Null FROM T_Numbers WHERE Num=0
UNION
SELECT True FROM T_Numbers WHERE Num=0
UNION
SELECT False FROM T_Numbers WHERE Num=0
```

Notice however that the following Query, which is very similar, **does not crash**, so it is not easy to formulate in what cases a **Null** will crash a **Union** operation:

```
SELECT Null FROM T_Numbers WHERE Num=0
UNION
SELECT True FROM T_Numbers WHERE Num=0
```

- A **user-defined VBA** function that contained a data-type mismatch (in a variable assignment, operator or function argument).

You **fix** these problems by either (or both):

- Correcting a mishandled **Null**: click “K.5 Why and how should I carefully handle Nulls in my Queries?”.
- Preventing type-error “#Error” in your Query: click “J.11.10 How do I fix a Query producing “#Num!”, “#Div/0!”, “#Error”, “#Type!” or “#Func!” in a field?”.

The Table “T\_Numbers” used in the examples above is an auxiliary Table that just contains integer numbers (click K.2.2).

#### J.10.4 How do I fix the crash “**Data type mismatch.**”?

If this is a **test-and-proven Query**, the **most likely** cause for the crash is that **Table data is inconsistent**. You **fix** this by clicking “J.1 How do I fix an error/crash in a test-and-proven Query?”.

If this is a **non-test-and-proven Query**, the crash is most likely because of one (or more) of the following:

- A table-type “#Type!” exception-value is an argument to a **domain** or **SQL**

<sup>91</sup> If you run the Query from the “SQL Access Editor”, a type-error “#Error” exception-value as an argument to a **domain** aggregate function will produce a **different** error message: “Unknown.”.

<sup>92</sup> This example and the next one are included in the Query “J\_Union\_Nulls” from file “Company\_Database.accdb”.

aggregate function<sup>93</sup>.

- A **user-defined** VBA function that contained a data-type mismatch (in a variable assignment, operator or function argument).

You **fix** these problems by preventing table-type “#Type!” in your Query: click “[J.11.10 How do I fix a Query producing “#Num!”, “#Div/0!”, “#Error”, “#Type!” or “#Func!” in a field?](#)”.

### J.10.5 How do I **fix** the **crash** “Invalid Procedure Call or Argument.”?

If this is a **test-and-proven Query**, the **most likely** cause for the crash is that **Table data is inconsistent**. You **fix** this by clicking “[J.1 How do I fix an error/crash in a test-and-proven Query?](#)”.

If this is a **non-test-and-proven Query**, the crash is most likely because of one (or more) of the following:

- A table-function “#Func!” exception-value is an argument to a **domain** or **SQL aggregate function**.
- A table-function “#Func!” exception-value in **some Union** operations.

You **fix** these problems by preventing table-function “#Func!” in your Query: click “[J.11.10 How do I fix a Query producing “#Num!”, “#Div/0!”, “#Error”, “#Type!” or “#Func!” in a field?](#)”.

### J.10.6 How do I **fix** the **crash** “Invalid use of Null.”?

If this is a **test-and-proven Query**, the **most likely** cause for the crash is that **Table data is inconsistent**. You **fix** this by clicking “[J.1 How do I fix an error/crash in a test-and-proven Query?](#)”.

If this is a **non-test-and-proven Query**, the crash is most likely because **Null** was used as an argument to a function, when the data type of the function argument is **Variant** but the function cannot handle the **Null**. For example, the following Query will crash:

```
SELECT Cdbl(Null) FROM T_Numbers
```

You **fix** this problem by modifying your SQL code or Table data to guarantee that **Null** is not used as indicated above. On **many** cases the cause is a mishandled **Null**: click “[K.5 Why and how should I carefully handle Nulls in my Queries?](#)”.

The Table “T\_Numbers” used in the example above is an auxiliary Table that just contains integer numbers (click [K.2.2](#)).

### J.10.7 How do I **fix** the **crash** “The expression cannot be used in a calculated column.”?

This crash is most likely because a calculated-invalid “#Invalid” exception-value is an argument to an **SQL aggregate function**<sup>94</sup>.

<sup>93</sup> If you run the Query from the “SQL Access Editor”, a table-type “#Type!” exception-value as an argument to a **domain** aggregate function will produce a **different** error message: “Data type mismatch in criteria expression.”.

<sup>94</sup> If you run the Query from the “SQL Access Editor”, a table-invalid “#Invalid” exception-value as an

You **fix** this by clicking “*J.11.11 How do I fix a Query producing “#Invalid” or “#Deleted” in a field?*”.

### **J.10.8 How do I fix the crash “Cannot have ... in aggregate argument.”?**

If a Query crashes with error message:

*“Cannot have Memo, OLE, or Hyperlink Object fields in aggregate argument (Val\_exp).”*

where “*Val\_exp*” stands for the expression used as argument to the aggregate function that caused the crash.

The crash is because you used an **aggregate** function other than “**Count ()**” or “**DCount()**” over a **Memo, OLE, or Hyperlink Object** argument.

You **fix** this by modifying your SQL code to guarantee that aggregate functions other than “**Count ()**” or “**DCount()**” are never used over a **Memo, OLE, or Hyperlink Object** argument.

### **J.10.9 How do I fix the crash “Could not find field...”?**

If a Query crashes with error message:

*“Could not find field 'Field\_name’”*

this crash is most likely because the “**Expression**” property of a **Calculated** field or Table validation rule uses the Table field “*Field\_name*” that does not exist.

You **fix** this by **finding** the corresponding Table and doing either of the following:

- **Correct** the corresponding **Calculated** field(s) and/or the Table’s record validation rule so they refer to existing field names.
- **Add** a field (creating or renaming it) with the same field name as the missing one.
- **Remove** the corresponding **Calculated** field(s) and/or Table’s record validation rule. This is risky, so you have to be very sure that this is what you want.

To **find which Table** produced the error notice that:

- You recently **removed** the field “*Field\_name*” from it.
- **It is used** (directly or indirectly through auxiliary Queries) in this Query.
- It has **Calculated** fields and/or a record validation rule.

### **J.10.10 How do I fix the crash “The expression cannot be used in a Calculated column”?**

This crash is most likely because the “**Expression**” property of a **Calculated** field is either **wrong** or it uses a Table field that was deleted.

You **fix** this by **finding** the Table and doing either of the following:

- **Correct** the corresponding **Calculated** field(s) so they refer to existing field names.
- **Add** a field (creating or renaming it) with the same field name as the missing one.

---

argument to a **domain** aggregate function will produce a **different** error message: “*Unknown.*”.

- **Remove** the corresponding **Calculated** field(s). This is risky, so you have to be very sure this is what you want.

To **find which Table** produced the error notice that:

- You recently **removed** some field(s) from it.
- **It is used** (directly or indirectly through auxiliary Queries) in this Query.
- It has **Calculated** fields.

#### **J.10.11 How do I fix the crash “Expression is too complex.”?**

This crash is most likely because an expression is too complex to be executed. For example, you have too many nested functions (e.g., “**Iif()**”, “**Switch()**” or others) and/or you have lots of combined value operators.

Notice however, that this crash **may also be** from a syntax error that causes a wrong interpretation of the expression.

If the crash cause is a syntax error, you **fix** it debugging your Query (click *J.7*) until you find the syntax error.

If the crash cause is from the complexity of the expression, you **fix** it simplifying it. For example, you may create a user-defined VBA function, or you may modify your SQL code so the calculation the expression does is divided in two steps.

#### **J.10.12 How do I fix the crash “Query is too complex.”?**

This crash is most likely because an SQL operation is too complex to be executed. For example, you have too many nested **Union** operations, or too many **Join** operations. In the trials I did, when I connected more than 50 SQL operations using 49 “**UNION**” operator the Query crashed.

Notice however, that this crash **may also be** from a syntax error that causes a wrong interpretation of the Query.

If the crash cause is a syntax error, you **fix** it debugging your Query (click *J.7*) until you find the syntax error.

If the crash cause is from the complexity of the Query, you **fix** it simplifying the Query by moving part of its SQL operation to one (or more) auxiliary Queries and invoke these auxiliary Queries from the main Query.

#### **J.10.13 How do I fix the crash “Cannot open any more databases.”?**

This crash is most likely because you have too many opened Queries. This is particularly likely when you are working with linked Tables.

You **fix** this by closing some of the Queries (click *B.4.I.7*).

If you want to know more about **linked** Tables, you may click “*K.3 How do I structure and optimize a distributed database?*”.

#### **J.10.14 How do I fix a crash from my user-defined VBA functions?**

If a Query **crash** opens the **VBA editor window** (click *K.9.1*), this is a crash from a user-defined VBA function. The VBA editor will open in the **specific line of code** that caused the crash. This will allow you to know what user-defined function caused the

crash.

If this is a **test-and-proven VBA function**, the **most likely** cause for the crash is that **Table data is inconsistent**. If you want some guidance on locating errors in your Table data, you may click "[J.1 How do I fix an error/crash in a test-and-proven Query?](#)".

Otherwise, if this is a **non-test-and-proven VBA function** you will have to debug your user-defined VBA function. VBA debugging is out of the scope of this **Lightning Guide**.

## J.11 How do I fix defective Query results?

When I say that results are **defective** it is not that they are **different** from what you intended, but rather that they are actually **flawed** (click [J.7.3](#)). You **fix** this by finding the **specific** problem you have in the sections below, and clicking on the corresponding section:

- "[J.11.1 How do I fix a Query showing a wrong numeric-like value?](#)"
- "[J.11.2 How do I fix a Query showing a wrong Short Text value?](#)"
- "[J.11.3 How do I fix a Query erroneously considering two different values as equal?](#)"
- "[J.11.4 How do I fix a Query erroneously considering two equal values as different?](#)"
- "[J.11.5 How do I fix a Query erroneously ordering records?](#)"
- "[J.11.6 How do I fix a Query requesting a non-declared parameter?](#)"
- "[J.11.7 How do I fix a Query requesting a non-existing parameter?](#)"
- "[J.11.8 How do I fix a Query requesting the same parameter twice \(or more times\)?](#)"
- "[J.11.9 How do I fix a Query showing "#####" in a field?](#)"
- "[J.11.10 How do I fix a Query producing "#Num!", "#Div/0!", "#Error", "#Type!" or "#Func!" in a field?](#)"
- "[J.11.11 How do I fix a Query producing "#Invalid" or "#Deleted" in a field?](#)"
- "[J.11.12 How do I fix a Query showing a black square in a field?](#)"
- "[J.11.13 How do I fix a Query producing "#Name?" in a field?](#)"
- "[J.11.14 How do I fix a Query showing a blank field that should not be blank?](#)"
- "[J.11.15 How do I fix a Query producing defective values?](#)"
- "[J.11.16 How do I fix a Query's defective "ON" expression?](#)"
- "[J.11.17 How do I fix a Transform Query producing wrong field names?](#)"
- "[J.11.18 How do I fix a Query that stalls/freezes?](#)"
- "[J.11.19 How do I fix a Query making arithmetic errors?](#)"
- "[J.11.20 How do I fix a Query making rounding errors?](#)"
- "[J.11.21 How do I fix a Query that I cannot open in "Design View"?](#)"



- “J.11.22 How do I fix my VBA functions comparing text strings case sensitive?”

### **J.11.1 How do I fix a Query showing a wrong numeric-like value?**

If you see a **numeric-like** value that is **different** from the **actual value** that the Query is producing, this is most likely because the **formatting** configured for this field and/or the field **width** and **height** is/are not the one you want. This is particularly frequent with **Date/Time** fields, where you can configure a format that only shows the **date-part** or the **time-part** of the stored value, and therefore, the actual stored value is not correctly shown.

You **fix** this by changing the field formatting and/or the field width and height to more suitable ones.

If you want to know more about how to change the field formatting, you may click:

- “H.6 How do I configure the formatting of column values in a Table/Query/Form?”

If you want to know more about how to change the field width and height, you may click:

- “H.1 How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?”
- “H.2 How do I change row height, hide rows or change row order in a Table/Query/Form?”

### **J.11.2 How do I fix a Query showing a wrong Short Text value?**

If you see a **Short Text** value that is **different** from the **actual value** that the Query is producing, this is most likely because the string contains invisible characters or because the field width and height is not large enough.

You **fix** this by either fixing the Query to suppress the invisible characters (if they are wrong) or by making the field width and/or height larger.

If you want to know how to suppress the invisible characters, you may click:

- “L.7.5 How do I fix text strings and invisible characters?”.

If you want to know how to make the field width and/or height larger, you may click:

- “H.1 How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?”
- “H.2 How do I change row height, hide rows or change row order in a Table/Query/Form?”

### **J.11.3 How do I fix a Query erroneously considering two different values as equal?**

This is most likely because one or more of the following causes:

- **Short Text or String values: case insensitiveness**  
MS-Access comparison operators consider an upper-case letter and its corresponding lower-case letter as the same one. For example, the text strings “Hello”, “HELLO” and “hello” are considered to be the same, even though they look different to you.

If you want to know more about this, you may click [L.7](#).

- **Same data/field type: different formatting**

If you have formatted two values (e.g., from two different fields) to be displayed in a different way, the same value may look different. You may then consider they are different, while MS-Access is correctly considering them as equal. For example, a *True Boolean* value may be displayed as **True**, **Yes** or **-1**. This may lead you to believe that they are different things, but it is the same value. A given **Date/Time** value (e.g., **3-Jan-2020 5:25:00**) may be displayed in different ways because the corresponding fields have different formatting (e.g., **3-Jan-2020 5:25:00**, **5:25:00** or **3-Jan-2020**), but it is actually the same value. The same happens with a given **numeric** value, that can be displayed differently (e.g., with exponential notation) even though it is the same value.

If you want to know more about formatting field values, you may click “[H.6 How do I configure the formatting of column values in a Table/Query/Form?](#)”.

Formatting problems not only apply to field values, and also apply to constants. Notice that a given value may be written as a constant in many different ways. If you want to know more about writing constants, you may click “[G.4 How do I write a constant?](#)”.

- **Different data/field types: type equivalences**

You may think that values from different data/field types should be different, but this is not so. For example, you may think that *True* is clearly different from -1, but in fact they are the same. Also, you may think that a **Date/Time** or *Date* value is clearly different from a **numeric** data/field type value, but they may actually be the same.

If you want to know more about this, you may click “[G.2.3 What is the result of combining different data/field types in expressions?](#)”.

Type equivalence problems not only apply to field values, and also apply to constants. Notice that a given value may be written as a constant in many different ways. If you want to know more about writing constants, you may click “[G.4 How do I write a constant?](#)”.

Notice that in principle there is **nothing to fix**, it is just that you are not interpreting correctly the way MS-Access **displays** and **stores** values. However, maybe you actually need to **fix** the database configuration or Query code, so you get the desired results.

#### **J.11.4 How do I fix a Query erroneously considering two equal values as different?**

This is most likely because of one or more of the following causes:

- **Short Text or *String* values: field width/height and invisible characters**

Text strings values may be **shown differently** to what is actually **stored** in the field. The reason is that the field width/height may not allow to show the full text string stored in the field and/or that the text string contains invisible characters. This may lead you to believe that two fields contain the same text string, when they actually contain two different text strings.

If you want to know more about this, you may click “[L.7.1 Why text strings can be](#)

*different from what is shown?*”.

- **Numeric values: field formatting**

Numeric values may be **shown differently** to what is actually **stored** in the field. The reason is that the field formatting does not allow to show the full value, for example, not all decimal figures are shown. This may lead you to believe that two fields contain the same numeric value, when they actually contain two different ones.

If you want to know more about this, you may click “[J.11.1 How do I fix a Query showing a wrong numeric-like value?](#)”.

- **Date/Time or Date values: field formatting**

**Date/Time** or **Date** values may be **shown differently** to what is actually **stored** in the field. The reason is that the field formatting does not allow to show the full value, for example, a **Date/Time** field is formatted to show only the date-part or only the time-part. This may lead you to believe that two fields contain the same **Date/Time** value, when they actually contain two different ones.

If you want to know more about this, you may click “[J.11.1 How do I fix a Query showing a wrong numeric-like value?](#)”.

- **Short Text or String values vs. other values**

A **Short Text** or **String** value may be displayed in exactly the same way as a value from another data/field type, but in fact they are different values with totally different internal representation. For example, the **Short Text** or **String** value “**8-January-2020**” is **shown** exactly the same as the **Date/Time** or **Date** value “**8-January-2020**”. However, they are different values, and their internal representation is completely different (click [G.2](#)). The same applies to the **Short Text** or **String** value “**834**” and the **Number-Integer** or **Integer** value “**834**”.

If you want to know more about this, you may click “[L.7 How do I fix errors with Short Text or String fields?](#)”.

Notice that in principle there is **nothing to fix**, it is just that you are not interpreting correctly the way MS-Access **displays** and **stores** values. However, maybe you actually need to **fix** the database configuration or Query code, so you get the desired results.

### **J.11.5 How do I fix a Query erroneously ordering records?**

This is most likely because the values you **see** in the Query fields are **different from** what it is actually stored in them. You may check the causes for this in the previous two sections:

- “[J.11.3 How do I fix a Query erroneously considering two different values as equal?](#)”
- “[J.11.4 How do I fix a Query erroneously considering two equal values as different?](#)”

### **J.11.6 How do I fix a Query requesting a non-declared parameter?**

This case is when the Query requests a parameter that you **did not declare** using the “**PARAMETERS**” clause (click [F.12](#)), but that is a **valid**, although **undefined**, variable **name** (in the Query itself or within an auxiliary Query). This is **different** from a **non-**

**existing** parameter (click *J.11.7*).

You can have an **undefined** variable because you mistyped it, or because you thought it is a field of your input record-list, and it is not, or because you were doing cut/paste of SQL code and you brought in an undefined variable, or for many other reasons.

There is nothing specifically wrong with this, because MS-Access considers all undefined variables as parameters and will prompt you to enter their value when running the SQL operation. However, if this is not what you intended, it should be fixed.

You **fix** this by **searching** your SQL code for the exact name of each unintended requested parameter, **locating** it in the code, **finding** out why is it not defined, and **correcting** it.

### **J.11.7 How do I fix a Query requesting a non-existing parameter?**

This case is when the Query requests a **parameter** that:

- You **did not declare** using the “**PARAMETERS**” clause (click *F.12*), and
- it is **not a valid variable name** within the Query.

For example, if running the SQL operation, it requests a parameter “**requested\_name**”, and you search for “**requested\_name**” in the SQL operation and it does not exist. Also, the SQL operation either **does not** contain auxiliary Queries or the ones it uses also do not contain the variable “**requested\_name**”. This is **different** from a **non-declared** parameter (click *J.11.6*).

This is most likely because you are using a foreign-language version of MS-Access and it **translates** some **field names** in the Query SQL code!! Furthermore, if you are using the “**Access SQL Editor**”, the SQL code in the “**Access SQL Editor**” will remain **unchanged**, and if you check the Query code you will still see the English field names there. This may be an extremely puzzling problem if you are not aware of this.

This problem happens in foreign-language versions of MS-Access when you are using a variable (e.g., a Table field) name that has an English meaning like “**Quarter**”. If you have a Query that uses field name “**Quarter**”, and you modify the Query design in “**Design View**” (for example selecting ascending order), MS-Access will **replace** in the SQL code the field name “**Quarter**” by the MS-Access translated name (in Spanish, it is replaced by “**Trimestre**”). When you then run the Query, it will request parameter “**Trimestre**”, because it is a field name that is used in the Query (because MS-Access **changed** it) but it **does not exist** in the Table.

I did a few trials and so far I have seen that this affects English variable names “**Year**”, “**Quarter**”, “**Month**”, “**Day**” and “**Trim**”, that are all translated to the foreign-language.


This may be an MS-Access **bug**.

If you are using a foreign-language version of MS-Access, you may click “*L.8.12 How do I fix foreign-language issues of MS-Access?*”.

### **J.11.8 How do I fix a Query requesting the same parameter twice (or more times)?**

This happens because you **saved** the Query **after** having manually selected record sorting on the Query “**Datasheet View**” or because you explicitly configured record

ordering in “**Datasheet View**”. This should not happen: this may be an MS-Access **bug**. You **fix** this by clearing the “**Order\_by**” property of the “Query properties”, following the steps:

1. Open the Query in “**Design View**” (click *B.4.1.3*).
2. Either click on the **Property Sheet**  icon from the “**Query Tools / Design**” contextual Ribbon, or rather, right-click anywhere on the “Query pane”, and then click on “**Properties**” from the pop-up menu. This will show the “**Property Sheet**” on the right side of the “Query pane”.
3. Click on the gray background of the top sub-pane, or rather, click on the white background of the bottom sub-pane, or else, click anywhere inside a column that does not correspond to any output field. This will show the “Query properties” in the “**Property Sheet**”.
4. In the “**Property Sheet**”, locate the row named “**Order\_by**” and clear the cell to its right.

To **prevent** that the same parameter is requested twice, **never** save the Query layout after having manually configured record sorting in Query “**Datasheet View**” and **never** explicitly configure record ordering in “**Datasheet View**”. If you want the Query to have some specific record ordering, do it by modifying the “**ORDER BY**” clause of its SQL Query code.

### **J.11.9 How do I fix a Query showing “#####” in a field?**

This is most likely because the **formatted** representation of that field’s value **does not fit** in that field’s cell size. This is, the formatted representation of the value is larger than the cell’s width and height, and therefore, the formatted value cannot be displayed in full in the cell.

When this happens, MS-Access does **not** show the value and will instead show the field **filled** with “#” characters (e.g., “#####”). Notice that this applies to all **numeric-like** data/field types, but not to the **Short Text** or *String* values. For the case of the **Short Text** or *String* data/field types, if the field’s width and height is not enough to show the complete value of the field, MS-Access will just show the leftmost part of the text string that fits in the field’s width and height.

You **fix** this by making the field width/height larger (click *H.1.1* or *H.2.1*) and/or modifying the field formatting (click *H.6*), so all the values do fit in the field size.

### **J.11.10 How do I fix a Query producing “#Num!”, “#Div/0!”, “#Error”**, “#Type!” or “#Func!” in a field?

If you see one of the following in a **Query** result field:

**#Num!**

**#Div/0!**

**#Type!**

**#Error!**

**#Func!**

this is most likely because either (or both):

- The expression that calculates the Query field's value **includes** a **Table field** that **contains** the exception-value “#Num!”, “#Div/0!”, “Type!” or “#Func!”. Remind that the type-error “#Error” exception-value **cannot** appear in a Table field, and rather the table-type “#Type!” exception-value is shown in case a type-error arises.
- The expression, **itself**, that calculates the Query field's value **is producing** the exception-value “#Num!”, “#Div/0!” or “#Error”.

In the first case, you **fix** this by correcting the corresponding Table error, clicking “[L.5.2 How do I fix a Table/Form showing “#Num!”, “#Div/0!”, “#Type!” or “#Func!” in a field?](#)”

In the second case, the cause of the error depends on the specific exception-value shown:

- If “#Num!” is shown, the expression resulted in the exception-value number-overflow. This happens when an operation result is **out of range** (e.g., 36000+45000 in an *Integer*), or when dividing zero by zero (e.g., “0/0” or “0 Mod 0”).
- If “#Div/0!” is shown, the expression resulted in the exception-value divide-by-zero. This happens when dividing a non-zero number by 0 (e.g., “8\0”).
- If “#Error” is shown, the expression resulted in the exception-value type-error. This happens when the data types of operands are not correct (e.g., multiply a number and a text string: “'text '\*5”).

You **fix** this by correcting the expression of the field and/or by correcting the SQL code, so no exception-value is produced. On **many** cases the cause is a mishandled **Null**: click “[K.5 Why and how should I carefully handle Nulls in my Queries?](#)”.

Notice that many (but not all) the expressions that result in an exception-value (e.g., “#Error”, “#Div/0!”, “#Num!”, ...) **will not crash the Query**, and the Query will work normally just showing the corresponding exception-value in some field(s). This implies that these errors can be passed on to other Queries. Therefore, consider that the cause of the error may not be in the Query where you first see it, and rather it can come from an auxiliary Query invoked from this one.

If you want to know more about exception-values, you may click “[J.15 What exception-value bugs can I get?](#)”.

Non-English MS-Access versions can show exception-values with a different (translated) text. For example, the Spanish version shows “#iNúm!”, “#iDiv/0!” and “#iFunción!”.

### **J.11.11 How do I fix a Query producing “#Invalid” or “#Deleted” in a field?**

If you see one of the following in a **Query** result field:


**#Invalid**

**#Deleted**

this is most likely because the expression that calculates the Query field's value **includes** a **Table field** that contains the exception-value “#Invalid” or “#Deleted”.

You **fix** this by correcting the corresponding Table error, checking:

- “L.5.3 How do I fix a Table/Form showing “#Invalid” in a field?”
- “L.5.4 How do I fix a Table/Form showing “#Deleted” in a field?”

For the specific case of “#Deleted”, this may be fixed by either clicking on the **Refresh All** “” icon from the “**Home**” Ribbon, or by closing the Query and running it again.

Non-English MS-Access versions can show a different (translated) text. For example, the Spanish version shows “#¡Tipo!”, “#Inválido”, and “#Nombre?”.

### J.11.12 How do I fix a Query showing a black square in a field?

If you see a **black square** in a checkbox field, this is exactly the same problem, and has the same fix, as described for the “#Deleted” case in the previous section J.11.11. The only difference is that you see a black square for the specific case of a field with **Yes/No** field type that is configured to be shown as a **checkbox**. In this type of fields, MS-Access shows a **black square** instead of the label “#Deleted”.

### J.11.13 How do I fix a Query producing “#Name?” in a field?

This is most likely because an SQL operation is producing the name-error “#Name?” exception-value. One example of an SQL operation<sup>95</sup> producing it is:

```
SELECT Null AS Name FROM T_Numbers
UNION ALL
SELECT True AS Name FROM T_Numbers
UNION ALL
SELECT False AS Name FROM T_Numbers
```

You **fix** this by correcting the SQL operation that produced this exception-value.

Notice that many (but not all) expressions that result in an exception-value (e.g., “#Error”, “#Div/0!”, “#Num!”, ...) **will not crash the Query**, and the Query will work normally just showing the corresponding exception-value in some field(s). This implies that these errors can be passed on to other Queries. Therefore, consider that the cause of the error may not be in the Query where you first see it, and rather it can come from an auxiliary Query invoked from this one.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click K.2.2).

Non-English MS-Access versions can show a different (translated) text. For example, the Spanish version shows “#Nombre?” instead of “#Name?”.

### J.11.14 How do I fix a Query showing a blank field that should not be blank?

This is most likely because the Table field contains a wrong **Null** or because the Query field's expression is producing a wrong **Null**.

If the problem is a wrong **Null** in a **Table** field, you **fix** this by clicking “L.5.12 How do

---

<sup>95</sup> This example is included in Query “J\_Union\_Nulls” from file “Company\_Database.accdb”.

*I fix an apparent Null in a Table/Form Short Text field configured as “Required=Yes”?”.*

If the problem is the **Query** producing the wrong **Null**, you **fix** this by correcting your SQL Query code. If you want to know more about this, you may click “*K.5 Why and how should I carefully handle Nulls in my Queries?”.*

### **J.11.15 How do I fix a Query producing defective values?**

A Query may **seem** to produce defective **values** because of one or more of the following causes:

- **Date/time or numeric values: value formatting**

The formatted **datetime** or **numeric** value **displayed** is different from the **actually stored** value (e.g., you see **5** but the stored value is **5.67**). This may lead you to believe that the Query is producing a wrong result.

This is not an error as such, so maybe there is nothing to fix. If you want to fix it, you only need to change the formatting of the field. If you want to know more about this, you may click “*H.6 How do I configure the formatting of column values in a Table/Query/Form?”.*

- **String values: insufficient cell size**

If the cell size showing the field’s value is not large enough, then **only part** of the text string that fits in the cell will be shown (e.g., you see “**John**”, but the stored value is “**John Lennon**”). This may lead to you to believe that the Query is producing a wrong result.

This is not an error as such, so maybe there is nothing to fix. If you want to fix it, you only need to increase the cell width (click *H.1.1*) and/or height (click *H.2.1*), so the full text string is seen.

- **String values: invisible characters**

If the string contains invisible characters (e.g., line-feed, horizontal tab), what you see may be different from the actually stored string (e.g., you see “**John**” but the stored value is “**John<line-feed>Lennon**”). This may lead to you to believe that the Query is producing a wrong result.

This is not an error as such, so maybe there is nothing to fix. If you want to **fix** this, you only need to modify the values and/or expressions that produce this result. If you want to know more about this, you may click “*L.7 How do I fix errors with Short Text or String fields?”.*

- **Using the same name for an input field as the one of an output field**

If you are using the same name for an **input** field as the one of an **output** field, then you **must** qualify the **input** field name prefixing it with the name of the inner SQL operation. If you **forget** to do it, then you are referring to the **output** field, and this can create confusion. In the following example, the SQL code of the inner **Select** named “**Inner\_select**” contains an **output** field named “**Age**”. The outer **Select** wants to present the **input** field “**Age**” to the square and the **input** field “**Age**” plus



3. The SQL code<sup>96</sup> would be:

```
SELECT (Inner_select.Age^2) AS Age, Age+3 AS Age_plus_3
FROM
(
  SELECT Num AS Age FROM T_Numbers
) AS Inner_select
```

However, the value of “**Age\_plus\_3**” is **not** the value of the **input** field name “**Age**” plus **3**. Rather, it is the value of the **output** field name “**Age**” (i.e., the **input** field name “**Age**” to the square) plus 3.

You **fix** this by qualifying the **input** field name (prefixing to it the **name** assigned to the inner SQL operation), but this requires qualifying **all** such **input** names. My advice is that you use as **output** field name the **input** field name with the “\_” suffix. This can be alternated as a series of Queries (chains of unions, ...) to distinguish the **input** field names and the **output** field names in every Query. This is clearer, and you avoid having to qualify the **input** field name. The resulting code would be:

```
SELECT Age^2 AS Age_, Age+3 AS Age_plus_3
FROM
(
  SELECT Num AS Age FROM T_Numbers
) AS Inner_select
```

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

### **J.11.16 How do I fix a Query’s defective “ON” expression?**

If the “**ON**” *Boolean* expression contains a user-defined function, the cause may be that the function is not dealing properly with **Nulls** and/or exception-values.

You **fix** this by either:

- Correcting the user-defined function so it handles **Nulls** properly.
- Replacing the function with an equivalent expression.
- Doing some workaround.

If the “**ON**” *Boolean* expression does not contain a user-defined function the problem may come from case insensitiveness, **Nulls** or exception values.

If you want to know more about **Nulls**, you may click “[J.14 What Null-related bugs can I get?](#)” and “[K.5.4 How do I handle Nulls in my Queries?](#)”.

If you want to know more about exception-values, you may click “[J.15 What exception-value bugs can I get?](#)”.

### **J.11.17 How do I fix a Transform Query producing wrong field names?**

In foreign-language versions of MS-Access the built-in conversion from **numbers** to **strings** used by the “**PIVOT**” clause may be different to what you expect because it will use the English decimal separation convention. This is, it will convert the decimal separation character to “\_”, instead of to the character used in your country (e.g., a

---

<sup>96</sup> This example and the next one are included in the Query “**J\_Field\_names**” from file “**Company\_Database.accdb**”.

comma “,”) because this is what it does in the English version (it considers that the decimal separation character is a **period** “.”).

You fix this by enclosing the numeric “**PIVOT**” **expression** in the “CStr()” type conversion function. Notice that the “CStr()” function **correctly** changes the decimal separation character to whatever character is used in the foreign-language.

Let me show you an example for the Spanish version of MS-Access. If the result of the “**PIVOT**” **expression** is the number “3.5” (which in the Spanish version would be “3,5”), the resulting **field name** would be “3\_5”. However, if you enclose the “**PIVOT**” **expression** in the “CStr()” function, the resulting **field name** would be “3,5”.

If you are using a foreign-language version of MS-Access, you may click “[L.8.12 How do I fix foreign-language issues of MS-Access?](#)”.

### J.11.18 How do I fix a Query that stalls/freezes?

You **fix** this pressing “**Ctrl-Pause/Interr**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**Pause/Interr**” key<sup>97</sup>).

This should interrupt the execution of the Query, and will show the error message:

*“Run-time error 3059:  
Operation cancelled by user.”*

in a dialogue-box with the buttons “**Continue**”, “**End**”, “**Debug**” and “**Help**”. You typically want to click “**End**” and continue working. The button “**Continue**” is shaded and cannot be clicked. The button “**Help**” usually offers very poor help. The button “**Debug**” will open the VBA editor: maybe you want to debug now, but most frequently you will want to do some checks on the data before jumping to debug your VBA code.

Notice that the case explained in this section, a Query that **stalls/freezes**, is very different from a Query **crash**. In a Query **crash** the Query does not run and you get an error message. If you want to know more about how to fix a Query **crash**, you may click “[J.9 How do I fix a crash from a syntax error?](#)” or “[J.10 How do I fix a crash from a run-time error?](#)”.

### J.11.19 How do I fix a Query making arithmetic errors?

Although you may not believe me at first, computers **systematically** make arithmetic errors. This happens because computers use **base 2** arithmetic, and computers have limitations in the number of bits they use to represent a value (which limits the precision and ranges of values they can store and process). Therefore, computers **systematically** make decimal/binary conversion rounding errors. If you want to verify this, you may click “[G.9.3 What are decimal/binary conversion rounding errors?](#)”.

---

<sup>97</sup> Notice that some laptop keyboards **do not have** the “**Pause/Interr**” key. You should then check in the laptop information what is equivalent to simultaneously pressing the “**Ctrl**” and “**Pause/Interr**” keys: in some laptops, it is simultaneously pressing the “**Ctrl**” and “**Fn**” and “**b**” keys.

### How do I fix arithmetic errors in comparison operators?

Binary/decimal conversion rounding errors can lead to logical errors in your comparison operators. For example, the following Query<sup>98</sup>:

```
SELECT Iif(5/10 - 4/10 - 1/10 = 0, "CORRECT", "WRONG!") AS Result
FROM T_Numbers WHERE Num=1
```

returns the result “**WRONG!**” because of decimal/binary rounding errors.

You **fix** this by introducing rounding expressions. For example, the Query above may be fixed using **either** of the two following rounding expressions:

```
SELECT Iif(abs(5/10 - 4/10 - 1/10) < 10e-15, "CORRECT", "WRONG!") AS Result
FROM T_Numbers WHERE Num=1
```

OR

```
SELECT Iif(Round(5/10 - 4/10 - 1/10,15) = 0, "CORRECT", "WRONG!") AS Result
FROM T_Numbers WHERE Num=1
```

The first one introduces a corrective rounding by taking the absolute value and comparing with  $10^{-15}$ . The second one introduces an explicit rounding function to 15 decimal digits.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click *K.2.2*).

### How do I fix arithmetic errors in stored values?

Binary/decimal conversion rounding errors can lead to logical errors in stored values. For example, if you introduce the value “**2147483583**” it will be stored as “**2147483520**” in a **Number-Integer** field.

You **fix** this by using a **Number-Double** field type-size instead of **Number-Integer**. In case you were already using a **Number-Double** field type-size, then the fix would require using the **Decimal** field/type size, using **Currency** field type or adjusting your database design. This is out of the scope of this **Lightning Guide**.

### J.11.20 How do I fix a Query making rounding errors?

When you want an **integer** result from a **non-integer** calculation, you have to **carefully** use a **rounding** function.

For example, if you want the number of trucks you need to transport the goods from a given order, the result must be integer, but the calculation will yield a non-integer number. If each truck has a capacity of 4,000 gallons, milk boxes are 23 gallons and water boxes are 45 gallons (and you can always fit them perfectly), the formula would be:

```
Num_trucks = Round(4000 / (23*Milk_boxes + 45*Water_boxes))
```

However, it is **not obvious** what “**Round ()**” should do. In principle, you could think “**Round ()**” should round **upwards**, because otherwise you will not be able to transport the full order. However, imagine that you have a policy that you only use full trucks, and the remaining load you serve with smaller vans. Then, “**Round ()**” should round

---

<sup>98</sup> The following three examples are included in Query “J\_Arithmetic\_Errors” from file “Company\_Database.accdb”.

**downwards.** Also, it could very well be that your policy is that you only serve with trucks as long as they are at least half-full, and the remaining load you serve with smaller vans. In this case, “**Round ()**” should round to the nearest integer.

The problem is also more general, because you may want to round to a certain number of decimals, and not only to integer values. For example, if you are calculating the monthly salary, you may want to round to two decimals (i.e., round to cents) and not to an integer number. The problem becomes even more general because you may want to round **positive** and **negative** numbers.

I suggest you check the **excellent** explanation about different rounding types from:

[en.wikipedia.org/wiki/Rounding](http://en.wikipedia.org/wiki/Rounding)

For your convenience, I am including in this **Lightning Guide** the following two tables (see the two next pages). The first table shows a **summary of frequent rounding types**. The second table shows a **summary of main differences** between them.

You should also be aware what is the type of rounding implemented by the different existing built-in functions of MS-Access:

- **Round(X, +d)**: Round-half to even
- **Int(X)**: Round down
- **Fix(X)**: Round towards-zero

Since you may frequently use Excel in combination with MS-Access, it is very convenient that you know the type of rounding implemented by the different existing built-in functions of Excel:

- **Round(X, d)**: Round-half away from zero

As you may see, the “**Round()**” function in MS-Access and in Excel implement **two different types** of rounding, so be careful with this.

In the lines above “**X**” represents the number to be rounded and “**d**” the number of decimal digits to which the number “**X**” should be rounded to. When “**d**” can only be a positive number, it is indicated as “**+d**”.

			Examples of numbers to round, and the corresponding result (rounding to zero decimals)																						
Decimal digit rounded	Amount Rounded	Direction of rounding	0	0.1	-0.1	0.5	-0.5	0.8	-0.8	3.1	-3.1	3.5	-3.5	3.8	-3.8	4	-4	4.1	-4.1	4.5	-4.5	4.8	-4.8		
<p>You may round to whatever <b>number of decimal digits</b> that you want. It is possible to round to thousands, hundreds, tens, integers, decimal, centesimal, ... Some functions have a second argument indicating the decimal digit to which you want to round: zero means round to units, 1 to 1/10, 2, to 1/100 and so on. Some functions accept negative second argument: in this case, -1 means round to tens, -2 round to hundreds, -3 round to thousands, and so on.</p>	<p><b>ROUND:</b> Also called "<b>directed rounding</b>". Value is rounded to the NEAREST integer decimal digit IN ONE DIRECTION. The direction towards the nearest integer decimal digit is taken is indicated by the rounding type. At most it changes a <b>whole</b> decimal digit.</p>	towards zero (or truncate, or away from infinity)	0	0	0	0	0	0	0	3	-3	3	-3	3	-3	4	-4	4	-4	4	-4	4	-4		
		away from zero (or towards infinity)	0	1	-1	1	-1	1	-1	4	-4	4	-4	4	-4	4	-4	5	-5	5	-5	5	-5		
		up (or ceiling, or towards plus infinity)	0	1	0	1	0	1	0	4	-3	4	-3	4	-3	4	-4	5	-4	5	-4	5	-4		
		down (or floor, or towards minus infinity)	0	0	-1	0	-1	0	-1	3	-4	3	-4	3	-4	4	-4	4	-5	4	-5	4	-5		
	<b>Amount Rounded</b>	<b>Tie breaking rule for .5 values</b>	<b>0</b>	<b>0.1</b>	<b>-0.1</b>	<b>0.5</b>	<b>-0.5</b>	<b>0.8</b>	<b>-0.8</b>	<b>3.1</b>	<b>-3.1</b>	<b>3.5</b>	<b>-3.5</b>	<b>3.8</b>	<b>-3.8</b>	<b>4</b>	<b>-4</b>	<b>4.1</b>	<b>-4.1</b>	<b>4.5</b>	<b>-4.5</b>	<b>4.8</b>	<b>-4.8</b>		
	<p><b>ROUND HALF:</b> Value is rounded to NEAREST integer decimal digit. If value is exactly <b>.5</b> decimal digit, then, you apply the <b>tie-breaking</b> rule indicated by the rounding type. At most it changes <b>half a decimal</b> digit.</p>	towards zero	0	0	0	0	0	1	-1	3	-3	3	-3	4	-4	4	-4	4	-4	4	-4	5	-5		
		away from zero	0	0	0	1	-1	1	-1	3	-3	4	-4	4	-4	4	-4	4	-4	5	-5	5	-5		
		up	0	0	0	1	0	1	-1	3	-3	4	-3	4	-4	4	-4	4	-4	5	-4	5	-5		
		down	0	0	0	0	-1	1	-1	3	-3	3	-4	4	-4	4	-4	4	-4	4	-4	4	-5	5	-5
		to even (or banker's rounding)	0	0	0	0	0	1	-1	3	-3	4	-4	4	-4	4	-4	4	-4	4	-4	4	-4	5	-5
			to odd	0	0	0	1	-1	1	-1	3	-3	3	-3	4	-4	4	-4	4	-4	5	-5	5	-5	

<b>Comparison of rounding techniques</b>	<b>Round up/down</b>	They accumulate rounding errors if you add positive numbers only, negative numbers only and also if you add both positive and negative numbers	
	<b>Round to/away from zero</b>	They cancel rounding errors if you add random positive and negative numbers, but, accumulate rounding errors if you add only positive numbers or only negative numbers	
	<b>Round half up/down</b>	All round-half functions cancel rounding errors on numbers with non-half decimal digits. For numbers with half decimal digits, see explanation to the right.	They accumulate rounding errors if you add positive numbers only, negative numbers only and also if you add both positive and negative numbers
	<b>Round half to/away from zero</b>		They cancel rounding errors if you add random positive and negative numbers, but, accumulate rounding errors if you add only positive numbers or only negative numbers
	<b>Round half to even/odd</b>		They cancel rounding errors if you add random positive numbers, or random negative numbers, or random positive and negative numbers.

### J.11.21 How do I fix a Query that I cannot open in “Design View”?

This is most likely because it is a **Union** Query, and **Union** Queries cannot be opened in “**Design View**”.

You **fix** this by enclosing the outermost **Union** operation of the Query in a **Select** operation. Remind to qualify all the fields from the enclosing **Select** operation with the identifier of the enclosed **Union** operation. If you want to know more, you may click “*K.4.7 Why should I enclose the outermost Union operation in a Select operation?*”.

### J.11.22 How do I fix my VBA functions comparing text strings case sensitive?

SQL comparison operators are all case insensitive (i.e., they consider each upper-case letter and its corresponding lower-case letter(s) as being the same. However, VBA functions and operators can be configured to handle string comparison in three different ways, depending on the setting of “**Option Compare**”, as follows:

- **Option Compare Database**  
This option can only be used with MS Access VBA. When set, uses the same setting as MS-Access to compare strings in VBA.
- **Option Compare Binary**  
This is the **default** VBA setting. When set, compares text at binary level, and therefore, it does it in a case sensitive way. Therefore, “HELLO” is considered **different** than “hello”.
- **Option Compare Text**  
When set, compares text in case insensitive way, so “HELLO” is considered **the same** as “hello”.

If VBA is comparing strings in binary or case sensitive way, it is due to the setting of “**Option Compare**”, as indicated above. I advise you to set “**Option Compare Database**” (click *D.10.4.4*), in order to have the same text string handling both in your SQL code and in your VBA code. In this way, you avoid errors arising from incoherence between SQL and VBA.

## J.12 What do I do when I just cannot fix a Query?

Sometimes you debug and debug, everything seems correct, and yet the Query fails.

It is possible that MS-Access has some internal restriction (e.g., number of nested Queries) that you are not aware of. It can also be that MS-Access has a bug. Regardless of the cause, do not get stuck and frustrated insisting on that MS-Access accepts your code.

The best approach is to try a **workaround** by changing the way your Query is written. Notice there are **many ways** to write a Query that yield **the same result**. I give you here some ideas for workarounds:

- **Change the name of the Query**  
Believe it or not, I had cases where a Query crashed, and changing its name made it work.

- **Copy the code to a new Query**  
Create a new Query and copy/paste all the Query code from the Query that does not work into the new Query. This may sound stupid, but sometimes a Query gets internally corrupted in MS-Access and doing this fixes the problem.
- **Split a Union Query in two**  
If the Query has an outermost **Union** operation, split it two different Queries, each of them having part of the outermost **Union** operation, and then create a third Query that is a **Union** of these two Queries.
- **Create auxiliary Query(es)**  
Take one (or more) SQL operation(s) from your Query, copy its/their code into a new auxiliary Query, and replace the SQL operation by an invocation of the auxiliary Query. This is particularly effective when you have a very large Query making use of none, or few, auxiliary Queries. Notice that this is a generalization of the previous suggestion of splitting a **Union** Query in two auxiliary Queries plus a main Query with a **Union** of both of them.
- **Remove auxiliary Query(es)**  
Replace the invocation of one (or more) auxiliary Query(es) by the SQL operation of the auxiliary Query. This is the reciprocal case from the previous bullet point.
- **Modify your expressions**  
Try to simplify your expressions using other operators (e.g., use a “**Switch()**” instead of nested “**If()**”, ...). Replace a very complex expression by a user-defined function. Introduce more SQL code instead of a very complex expression, by writing the Query in a different way.
- **Restructure your SQL code**  
Change the logic in which the Query is written. You can split the functionality in a series of smaller SQL operations, each computing part of the desired results, combining them with a **Union** operator. You can modify the way **Join** operations are nested. You can simplify the “**ON**” expression in **Joins** using a more complex “**WHERE**” expression.

Doing a **workaround** is extremely effective way of fixing Queries if you get completely stuck while debugging them.

### **J.13 Why should I always compare the results of an existing Query?**

If you are debugging a test-and-proven existing Query because you detected a bug in it, or because you are modifying its functionality, when you are done debugging, you should **always** compare the results of the **new** Query code with the results of the **old** Query code, over **the actual database data**.

To do this, **always** keep the old Query until you compare the results. You can efficiently compare the results of both Queries (the one with the old Query code and the one with the new Query code) using a **record-list** comparator tool (you may see *J.13.1* slightly further below).

Doing this will allow you to check **both** that the **new** behavior you expected in your new



Query code is correct (e.g., you have a new field), but **most important**, that the old behavior that **should still be there** is also correct (e.g., that the values produced in all the other fields are the same as previously). In other words, it is essential that you **always** check that your SQL code modifications to correct the bug or to improve the functionality **have not damaged** the pre-existing correct functionality. Notice that a test and proven Query is a **treasure** that you should be **very careful to preserve**.

As I indicated above, this comparison should be done over your real database data, not only with some test cases. The reason is that the real database data usually contains many cases that you had not expected, and you do not have in your test cases. It is therefore **much sounder** to check the results of your new Query code against your old Query code over the **real database data** than doing it over **some test data** that you have.

### **J.13.1 What is a record-list comparator tool?**

It is a tool that compares two record-lists and **marks all the differences** between them. It should also allow you to **easily** modify the two record-lists to focus the comparison on some records. The tool should therefore be capable of ordering the records/fields as you need, remove records/fields as you need, add records/fields as you need and edit records/fields as you need. Using such a tool, you can compare two record-lists and progressively modify them to focus at each point on the parts that are relevant for you.

For example, imagine that you did a Query modification to process invoices with VAT=20% in a different way. You then want to **compare** the results of the Query **before** and **after** the modification. Using the record-list comparator tool you **first** check that the results for all the other invoices (e.g., the ones that have VAT other than 20%) **are exactly the same** as previously. This is essential to check that you have not damaged the previous functionality. After having done this, you can remove from both record-lists all such invoices and focus on the results for the ones with VAT=20%. Here you will need to check that the results that you want to be the same are actually the same, and the ones that should be different are actually different and are different in the exact way you wanted. All of this is done with the support of the record-list comparator tool.

One way of accessing a simple but useful record-list comparator tool is building it using Excel. Excel has cell-reference formulas that allows you to build a file to compare the cells in two sheets based on the cell position (i.e., A1 compared with A1, and so on). You may think that you achieve this using absolute cell references (i.e., \$A\$1), but this is **wrong**. The reason is that you need that A1 in one sheet is compared with A1 in the other sheet even if you add/remove rows/columns and even if you reorder rows/columns. You can achieve this using a formula with cell-reference functions. Also, you need to compare string fields as strings (i.e., requesting equal match), but it is **extremely convenient** to compare numerical results allowing for some (configurable) error margin. The reason for needing some error margin is that when you modify a Query, the numerical results may not be exactly the same because of rounding errors in the calculations. In most cases you want to ignore such rounding errors and consider that the result 25.00000001 is the same as 24.999999999.

I have developed such a record-list comparator Excel file that you can download from the link:

[https://lightningguide.net/Record\\_List\\_Comparator.xlsx](https://lightningguide.net/Record_List_Comparator.xlsx)

## J.14 What Null-related bugs can I get?

You **fix** this by finding the **specific** bug in the sections below, and clicking on it:

- “J.14.1 What effect does Null cause in Query results?”
- “J.14.2 What effects does Null cause in Arithmetic, Comparison and Pattern value operators?”
- “J.14.3 What effects does Null cause in Logical value operators?”
- “J.14.4 What effects does Null cause in Text string value operators?”
- “J.14.5 What effects does Null cause in the “IN” and “NOT IN” Miscellaneous value operators?”
- “J.14.6 What effects does Null cause in SQL operators that remove duplicate records?”
- “J.14.7 What effect does Null cause in Union SQL operators?”
- “J.14.8 What effects does Null cause in aggregate functions?”
- “J.14.9 What effects does Null cause as an argument of a VBA function?”

### J.14.1 What effect does Null cause in Query results?

When a Query’s **outermost** “**SELECT**” **expression** returns **Null**, the corresponding Query record field will be **Null**. Each **Null** will be **shown** as an **empty field** in the corresponding records of the **output** record-list of the Query, same as it is shown in Table fields with **Null**.

### J.14.2 What effects does Null cause in Arithmetic, Comparison and Pattern value operators?

All Arithmetic, Comparison and Pattern value operators return **Null** when one (or both) operands is **Null**. Notice that this implies that “**Null = Null**” returns **Null** and **does not return True**: this is just one case, but it is an important one that you should remember well.

### J.14.3 What effects does Null cause in Logical value operators?

All Logical operators return **Null** when one (or both) operands is **Null**, **except** in the following cases:

- “False **AND** Null” and “Null **AND** False” return *False*.
- “True **OR** Null” and “Null **OR** True” return *True*.
- “Null **Imp** True” returns *True*.
- “False **Imp** Null” returns *True*.

Notice that the results in the cases above is what you would expect if **Null** is replaced by **unknown** or by a variable **X**.

#### J.14.4 What effects does Null cause in Text string value operators?

The “+” Text string operator returns **Null** when one (or both) operands is **Null**.

The “&” Text string operator returns **Null** when both operands are **Null**.

The “&” Text string operator returns **the other operand** when **one** operand is **Null** and the other operand is a valid text string.

#### J.14.5 What effects does Null cause in the “IN” and “NOT IN” Miscellaneous value operators?

If the **searched list** contains one or more **Null** they will be ignored and the operator will work normally. If the **searched value** is **Null**, both operators will return **Null**.

#### J.14.6 What effects does Null cause in SQL operators that remove duplicate records?

The “**UNION**” operator and **Select** operations using “**DISTINCT**” or “**DISTINCTROW**” consider two **Null** as **the same value** for the purpose of producing **distinct** records. If you have two or more records having the same values, or **Null**, in all their fields (comparing field by field) all of the said records are considered duplicate records, and only one of them is produced by the SQL operations above.

#### J.14.7 What effect does Null cause in Union SQL operators?

**Null** in **Union** operators can have different effects, like working well, producing an exception-value or crashing the Query. The best practice is avoiding **Null** in **Union** operations. Let me show you the following examples of the effect of **Null** in **Union** operations.

The following Query<sup>99</sup> produces the name-error “**#Name?**” exception-value:

```
SELECT Null FROM T_Numbers WHERE Num=0
UNION ALL
SELECT True FROM T_Numbers WHERE Num=0
UNION ALL
SELECT False FROM T_Numbers WHERE Num=0
```

The following Query (which is very similar) **works well**.

```
SELECT Null FROM T_Numbers WHERE Num=0
UNION ALL
SELECT True FROM T_Numbers WHERE Num=0
```

The following Query (which is very similar) **crashes**:

```
SELECT Null FROM T_Numbers WHERE Num=0
UNION
SELECT True FROM T_Numbers WHERE Num=0
UNION
SELECT False FROM T_Numbers WHERE Num=0
```

The following Query (which is very similar), does not crash, but in the second record it

---

<sup>99</sup> These four examples are included in the Query “**J\_Union\_Nulls**” from file “**Company\_Database.accdb**”.

produces a box with a question mark instead of the value True

```
SELECT Null FROM T_Numbers WHERE Num=0
UNION
SELECT True FROM T_Numbers WHERE Num=0
```

The Table “T\_Numbers” used in the examples above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

### J.14.8 What effects does Null cause in aggregate functions?

All aggregate functions (click [F.7.18.7](#)) **ignore records that produce Null** in the expression of the function argument, **except** “[Count\(\\*\)](#)”, “[DCount\(“\\*”\)](#)”, “[First\(\)](#)”, “[DFirst\(\)](#)”, “[Last\(\)](#)” and “[DLast\(\)](#)”.

“[Count\(\\*\)](#)” and “[DCount\(“\\*”\)](#)” do not have an argument consisting of an expression over the field records. For this reason, one or more **Null** in the record fields does not affect them.

“[First\(\)](#)”, “[DFirst\(\)](#)”, and “[DLookup\(\)](#)” return **Null** if that is the result of the expression over the **first** record in their **group** of input records.

“[Last\(\)](#)” and “[DLast\(\)](#)” return **Null** if that is the result of the expression over the **last** record in their **group** of input records.

All aggregate functions, except “[Count\(\\*\)](#)”, “[DCount\(“\\*”\)](#)”, “[Count\(\)](#)” and “[DCount\(\)](#)”, return **Null** if **all** the records in its **group** of input records produce **Null** in the argument expression, or, if the **group** of input records is empty. The four functions [Count\(\\*\)](#), “[DCount\(“\\*”\)](#)”, “[Count\(\)](#)” and “[DCount\(\)](#)” return the valid value “0” in that case.

The **two-value** aggregate functions “[StDev\(\)](#)”, “[DStDev\(\)](#)”, “[StDevP\(\)](#)”, “[DStDevP\(\)](#)”, “[Var\(\)](#)”, “[DVar\(\)](#)”, “[VarP\(\)](#)” and “[DVarP\(\)](#)” return **Null** if **group** of input records contains **zero or one** record(s) that produce a **non-Null** in the argument expression.

### J.14.9 What effects does Null cause as an argument of a VBA function?

Depending on the case, the VBA function may return a correct **non-Null** value, may return a **Null** or may **crash**. If the function crashes, the Query will also **crash**.

A **Null** as an argument of any **built-in** or **user-defined** VBA function, where the data type of the argument is **other than Variant**, will **crash** the Query showing the error message “*Data type mismatch in criteria expression.*”.

A **Null** as an argument of **most built-in** functions (e.g., “[Log\(\)](#)” or “[Sin\(\)](#)”) will **crash** the Query showing the error message “*Data type mismatch in criteria expression.*”.

A **Null** as an argument of any **built-in type conversion** function (“[CByte\(\)](#)”, “[CBool\(\)](#)”, ...) will **crash** the Query showing the error message “*Invalid use of Null.*”.

A **Null** as an argument of the “[IsNull\(\)](#)”, “[If\(\)](#)”, “[Nz\(\)](#)” or “[Switch\(\)](#)” **built-in** functions will **work correctly**. However, “[Nz\(\)](#)” and “[Switch\(\)](#)” evaluate the non-selected expression, so they are vulnerable to crashes. My advice is therefore that you use “[If\(\)](#)” and “[IsNull\(\)](#)” to handle **Nulls**. If you want to know more about this, you may click “[K.5.4 How do I handle Nulls in my Queries?](#)”.

A **Null** as an argument of a **user-defined** function will have the effect that is coded in the function. If the data type of the argument in the function definition is other than *Variant*, the function will **crash**. Therefore, whenever you expect that a function argument may be **Null** (which is **very** frequent), you should declare the argument data type as *Variant*, and you start the function code by checking if any of the arguments is **Null**. Depending on the existence of **Null** arguments, and the specific semantics of the function, you have to decide what the function should do. The most usual result of one, or more **Null** argument(s) is that your user-defined function will return **Null**, but this is not always the case.

## J.15 What exception-value bugs can I get?

You **fix** this by finding the **specific** bug in the sections below, and clicking on it:

- “J.15.1 What is an exception-value?”
- “J.15.2 What operations produce number-overflow “#Num!”?”
- “J.15.3 How do I prevent exception-values?”
- “J.15.4 What is the effect of an exception-value in value operators?”
- “J.15.5 What is the effect of an exception-value in function arguments?”
- “J.15.6 What is the effect of an exception-value in expressions?”
- “J.15.7 When is an exception-value crashing the Query?”

### J.15.1 What is an exception-value?

An **exception-value** is a **special result** that indicates that a **severe problem** was encountered when evaluating a **value operator**, an **SQL operator**, a **function** or a **value-reference**.

The following is a list of the **exception-values** you can get, and their most frequent causes:

- “**Number-overflow**” (shown as “#Num!”)
 

It is produced by a result that is **out of range** of the **output data type**, and also by dividing zero by zero (e.g., “0/0”, “0 Mod 0” or “0\0”). If you want to know more, you may click “J.15.2 What operations produce number-overflow “#Num!”?”.

Number-overflow “#Num!” appears mainly in value operators, functions and in **Calculated** Table fields.
- “**Divide-by-zero**” (shown as “#Div/0!”)
 

It is produced by dividing a non-zero by zero (e.g., “3/0”, “5 Mod 0” or “8\0”).

Divide-by-zero “#Div/0!” appears mainly in value operators, functions and in **Calculated** Table fields. If you want to know more, you may click “G.5 How do I use value operators in an expression?”.
- “**Type-error**” (shown as “#Error”)
 

It is produced by a data type mismatch in an operator or function (e.g., “'text'+5”). If you want to know more, you may click “G.5 How do I use value operators in an”.

*expression?*”.

Type-error “**#Error**” **cannot** be produced in **Calculated** Table fields.

- “**Table-type**” (shown as “**#Type!**”)
 

It is typically produced in a **Calculated** Table field when the calculating expression suffers a data type mismatch in an operator or function (e.g., “'text'+5”).

Table-type “**#Type!**” **cannot** be produced by value operators and functions.

- “**Table-function**” (shown as “**#Func!**”)
 

It is typically produced in a **Calculated** Table field when the calculating expression suffers a function crash (e.g., “Log('text')”).

- “**Table-invalid**” (shown as “**#Invalid**”)
 

It is produced in a **Calculated** Table field when the calculating expression makes a reference to a Table field that does not exist (e.g., “5\*Non\_existent\_field”).

Notice that when you save a Table design with such an erroneous **Calculated** field you get a warning to **prevent** this error (click *L.2.2*).

Table-invalid “**#Invalid**” **cannot** be **produced** in expressions, nor in SQL operations.

- “**Name-error**” (shown as “**#Name?**”)
 

It is produced in some SQL operations<sup>100</sup> like:

```
SELECT Null AS Name FROM T_Numbers
UNION ALL
SELECT True AS Name FROM T_Numbers
UNION ALL
SELECT False AS Name FROM T_Numbers
```

Name-error “**#Name?**” is not usually produced in **Calculated** Table fields, but it can be **shown** in **Form** fields, when the corresponding linked Table field does not exist. Name-error “**#Name?**” is not usually produced in value expressions.

These **exception-values** may either **crash** the Query where they appear, or they may **propagate** into other SQL operations and even to other Queries. In the case of them propagating, the Queries work normally, and they **return** the corresponding **exception-values** that are **shown** in the corresponding field(s) of the Query results.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click *K.2.2*).

Non-English MS-Access versions can show a different (translated) text for exception-values. For example, the Spanish version shows “**#iTipo!**”, “**#iNúm!**” or “**#iFunción!**”.

### J.15.2 What operations produce number-overflow “**#Num!**”?

An Arithmetic operation that causes **overflow** returns **number-overflow**. The operations that cause overflow are:

- **Adding, subtracting** or **multiplying** two *Byte*, *Integer* or *Long* values such that the result is out of range (positive or negative) of the *Long* data type.

<sup>100</sup> This is included in the Query “J\_Union\_Nulls” from file “Company\_Database.accdb”.

- **Adding, subtracting or multiplying** a *Single* value with another *Single* value, or with any **integer-like** value, such that the result is out of range (positive or negative) of the *Double* data type.
- **Adding, subtracting or multiplying** a plain *Double* constant (i.e., a *Double* constant not enclosed in a CDbI() function) with any **integer-like** or *Single* value such that the result is out of range (positive or negative) of the *Double* data type.
- **Adding, subtracting or multiplying** a *Variant-Decimal* value with any numeric data type value such that the result is out of range (positive or negative) of the *Variant-Decimal* data type.
- Raising a number to a **power** such that the result is out of range (positive or negative) of the **output** data type.
- **Dividing 0 by 0**: “0/0”, “0\0” or “0 Mod 0”.

Notice that concatenating text strings resulting in a text string longer than 255 characters **does not** produce overflow. Actually, you can handle extremely long text strings (longer than 3,000 characters) in your expressions.

### J.15.3 How do I prevent exception-values?

This section also answers the question:

- **Why cannot I handle exception-values?**

Unlike you do with **Nulls**, exception-values **cannot** be handled. Once an exception-value is produced, you **cannot** get rid of it. It will either be shown in your Query results or it will **crash** the Query.

This is so because there are no functions/operators that will produce a meaningful result when applied to an exception-value. In particular, the functions and operators you usually use to handle **Null do not** work:

- “**Iff()**” with an exception-value as first argument will either return the exception-value or crash the Query.
- Comparison operators (in particular “=” and “<>”) with an exception-value as an argument will either return the exception-value or crash the Query.
- “**Nz()**” with an exception-value as an argument will either return the exception-value or crash the Query.
- “**IsError()**” over an exception-value will either return the exception-value or crash the Query.
- “**IsNumeric()**”, “**IsDate()**”, “**IsArray()**” or “**IsEmpty()**” with an exception-value as an argument will either return the exception-value or crash the Query.
- “**Switch()**” with an exception-value as an argument will either return the exception-value or crash the Query.

Since exception-values cannot be **handled**, they have to be **prevented**.

In order to prevent exception-values in Queries, you should know how they are produced clicking “*J.15.1 What is an exception-value?*”. When you observe an exception-value in

one of your Tables or Queries, you have to take action to correct the error that is producing it.

A frequent cause for exception-values in Queries is a mishandled **Null**. If you want how to correct a mishandled **Null**, you may click “[K.5 Why and how should I carefully handle Nulls in my Queries?](#)”.

#### **J.15.4 What is the effect of an exception-value in value operators?**

Most exception-values as an operand of **all operators**, except “**IN**” and “**NOT IN**”, returns **the same exception-value**.

Most **exception-values** as an **operand** of operators “**IN**” or “**NOT IN**” will cause the following:

- If the exception-value is the **searched value**, the Query will **crash**.
- If the exception-value is one of the values in the searched list, and also, the searched value is **not** to its left in the list (i.e., the exception-value is reached **before** having found the searched value), the Query will **crash**.
- Otherwise, a correct value will be returned by the “**IN**” or “**NOT IN**” operators.

#### **J.15.5 What is the effect of an exception-value in function arguments?**

An **exception-value** as an argument of **most functions** will **crash** the Query. If you want to know more about this, you may click “[J.15.7 When is an exception-value crashing the Query?](#)”.

#### **J.15.6 What is the effect of an exception-value in expressions?**

If along the evaluation of an **expression**, an exception-value is produced in an operator other than “**IN**” and “**NOT IN**”, the expression will return that **first exception-value** that is produced.

If along the evaluation of an **expression**, an exception-value is produced as an operand of the “**IN**” or “**NOT IN**” operators or as an argument of a function, the expression will on most cases **crash**. If you want to know more about this, you may click “[J.15.7 When is an exception-value crashing the Query?](#)”.

If you want to know more about the evaluation order of expressions, you may click “[G.7 What is the evaluation order of an expression?](#)”.

#### **J.15.7 When is an exception-value crashing the Query?**

An **exception-value** will most likely **crash** the Query in the following cases:

- An **exception-value** as an **operand** of operators “**IN**” or “**NOT IN**”  
If the exception-value is the **searched value**, the Query will most likely **crash**.  
If the exception-value is one of the values in the searched list, and also, the searched value is **not** to its left in the list (i.e., the exception-value is reached before having found the searched value), the Query will most likely **crash**.
- An **exception-value** as an **argument** of **most built-in functions** and most user-defined functions will most likely **crash** the Query. In particular, an **exception-value** as an



argument of any type conversion function will most likely **crash** the Query.

An important case in which the **exception-value** does not **crash** the Query is when it is the argument to the “**Iif()**” function that is **not returned** (i.e., it is not evaluated): in this case, the “**Iif()**” function returns its other argument normally, but if the exception value is in the *Boolean* (first) argument, the Query will most likely crash.

- An **exception-value** in one, or more, record fields being processed by **any aggregate function** will most likely **crash** the Query. The exceptions are the “**Count (\*)**” and “**DCount("\*\*")**” aggregate functions that count the **records** in the group, regardless of their field values, and will return a correct value even if one or more record fields contain exception-values.
- If you do a **Union** with a Query that produces one or more **exception-values**, the **Union** operation will **crash**. The error message produced is the one of the **first** exception-value encountered.

When the Query crashes in the different cases above, the error message presented by MS-Access depends on the **exception-value** that **caused the crash**. If you want to know what are the different error messages and their causes, you may click “*J.10 How do I fix a crash from a run-time error?*”.

## J.16 What data type bugs can I get?

MS-Access uses a **weak** data type control. It does its best to interpret constants and field values involved in operators and functions as belonging compatible data types. This is very convenient, and is correct on most occasions, but it sometimes may lead to very puzzling Query behavior. The reason is that a value that you believe belongs to a given data type is interpreted by MS-Access as belonging to a different data type, yielding what seems to be a wrong result. For example, you probably think that the expression:

```
#1/1/2000# BETWEEN "1/1/1999" AND "1/1/2001"
```

returns **True**. However, this expression returns **False**, because the first *Date* value is converted to a text string with a different format, and the converted text string is not between the text strings "1/1/1999" and "1/1/2001".

If you want to know more, you may click:

- “*J.16.1 What data type bugs can I get with constants?*”
- “*J.16.2 What data type bugs can I get with value operators?*”
- “*J.16.3 What data type bugs can I get with the Union operator?*”
- “*J.16.4 What data type bugs can I get with aggregate functions?*”
- “*J.16.5 What data type bugs can I get with VBA functions?*”

### J.16.1 What data type bugs can I get with constants?

The data type of a constant will be decided by MS-Access depending on context. For example, the constant “**0**” can be interpreted as *Boolean*, *Long*, *Integer*, *Double*, *Single*, *Currency* or *String*. For example, the constant “**0**” may start as a number, but becomes a

*String* in the following Query<sup>101</sup> code:

```
SELECT 0 AS Field_1
FROM T_Numbers WHERE Num=1
UNION
SELECT "This is Text" AS Field_1
FROM T_Numbers WHERE Num=1
```

Whenever there is a possible data type ambiguity, my advice is you use a specific type conversion function over the constant. For example, “**CLng(0)**” to make sure that MS-Access will interpret it as the data type you want. If you want to know more about type conversion functions you may click “[G.2.5 How do I force a value to belong to a specific data type?](#)”.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

### **J.16.2 What data type bugs can I get with value operators?**

A very frequent data type problem with value operators is using a data type **not admitted** in an operator. For example, trying to multiply a number and a text string. The result is that the operator will return the **type-error** “#Error” exception-value.

Another frequent data type problem with value operators is getting an **apparently** wrong result because you **mistakenly** used a data type admitted in the operator, but that you did not want to use. For example, you may use a **datetime** value as an argument of Logical operators, but most likely you did not want to do it, and this is why you are getting an apparently wrong result.

In either case you **fix** this by changing the value that is creating the problem in the operator or by using a type conversion function (click [G.2.5](#)).

If you want to know what are the data types admitted for each value operator you may click “[G.5 How do I use value operators in an expression?](#)”.

In the following subsections I present more specific data type problems for each class of value operators:

- “[J.16.2.1 What data type problems can I get with Arithmetic operators?](#)”
- “[J.16.2.2 What data type problems can I get with Text string operators?](#)”
- “[J.16.2.3 What data type problems can I get with Comparison operators?](#)”
- “[J.16.2.4 What data type problems can I get with Pattern operators?](#)”
- “[J.16.2.5 What data type problems can I get with Logical operators?](#)”
- “[J.16.2.6 What data type problems can I get with Miscellaneous operators?](#)”

#### **J.16.2.1 What data type problems can I get with Arithmetic operators?**

You may get apparently wrong results when combining **datetime** values with **integer-like** or with **fractional** values to perform computations on dates and/or times. The most frequent cause for this problem is that you are not handling properly the numeric-like representation of **datetime** values. If you want to know more about this, you may click

---

<sup>101</sup> This is the Query “J\_Types\_Constant” from file “Company\_Database.accdb”.

“D.4.5 What is the “Date/Time” field type?”.

You may get apparently wrong results when combining **Boolean** or **Yes/No** values with **integer-like** or **fractional** values. The most frequent cause for this problem is that you may be **unaware** that **True/Yes/On** or **ticked** is converted to number “-1” while **False/No/Off** or **unticked** is converted to number “0”.

You may get apparently wrong results when using the “+” operator. The most frequent cause for this problem is that you are using it with text strings that represent numbers, and instead of adding them up it is concatenating both **string** operands.

### **J.16.2.2 What data type problems can I get with Text string operators?**

You may get apparently wrong results because of:

- The zero-length string, invisible characters or invisible strings: click “L.7.10 What apparently defective results can invisible characters produce?”.
- Type conversion from a **numeric-like** operand to a text string, which may be not obvious at all: click “G.5.2 What are the Text string operators?”.
- Leading or trailing spaces, other invisible characters or control characters, because they are **not** ignored by Text string operators and all of them are concatenated: click “L.7.10 What apparently defective results can invisible characters produce?”.

### **J.16.2.3 What data type problems can I get with Comparison operators?**

You may get apparently wrong results when comparing values of the same data/field type. The most frequent case for this problem is that you believe the values are being compared as a **numeric-like** data/field type, and rather, they are being compared as a **String** or **Short Text** data/field type.

You may get apparently wrong results when comparing **Boolean** or **Yes/No** values with **integer-like** or **fractional** values. The most frequent cause for this problem is that you may be unaware that **True/Yes/On** or **ticked** is converted to number “-1” while **False/No/Off** or **unticked** is converted to number “0”.

You may get apparently wrong results when comparing text strings. The most frequent causes for this problem are:

- The zero-length string, invisible characters or invisible strings: click “L.7.10 What apparently defective results can invisible characters produce?”.
- Type conversion from a **numeric-like** operand to a text string, which may be not obvious at all: click “G.5.2 What are the Text string operators?”.
- Leading or trailing spaces, other invisible characters or control characters, because they are **not** ignored by Text string operators and all of them are concatenated: click “L.7.10 What apparently defective results can invisible characters produce?”
- You are not aware that Comparison operators ignore the case (they are case insensitive).

- You are not aware that Comparison operators ignore leading spaces, but do not ignore trailing spaces, nor other invisible characters: click [L.7.10 What apparently defective results can invisible characters produce?](#).
- You are not aware of the ordering rules for text strings: click “[F.7.12.1 How are the different data/field types ordered by the “ORDER BY” clause?](#)”.

#### J.16.2.4 What data type problems can I get with Pattern operators?

You may get apparently wrong results because you are not handling properly the semantics of the pattern string. Click “[G.5.4 What are the Pattern operators?](#)”.

#### J.16.2.5 What data type problems can I get with Logical operators?

You may get apparently wrong results because one of the operands is **mistakenly** not a **Boolean** value. Notice that Logical operators work with operands of any **numeric-like**, and **string** data/field types, even if both operands are of different data/field types. Click “[G.5.5 What are the Logical \(Boolean\) operators?](#)”.

#### J.16.2.6 What data type problems can I get with Miscellaneous operators?

You may get apparently wrong results with “**IS IN**” or “**NOT IS IN**” operators because you are not handling properly the data types of the searched value and/or the values in the list.

You may get apparently wrong results with “**IS IN**” or “**NOT IS IN**” operators because of the same problems as comparison operators with text strings.

You may click “[G.5.7 What are the “IN” and “NOT IN” Miscellaneous operators?](#)”.

You may get apparently wrong results with “**BETWEEN AND**” or “**NOT BETWEEN AND**” operators because you are not handling properly the data types of the searched value and/or the values in the list. You may click “[G.5.8 What are the “BETWEEN AND” and “NOT BETWEEN AND” Miscellaneous operators?](#)”.

#### J.16.3 What data type bugs can I get with the Union operator?

MS-Access does not enforce **same-data-type** between the same fields of operands of the **Union** operator. As long as the **number of fields** is the same in the two operands, it will produce a result. MS-Access does not care if the data types of the fields are different, as long as the number of fields is the same.

I have made some tests and it seems MS-Access handles different data types in Union operations in the following way:

- The resulting data type from combinations of different **numeric-like** data/field types is the data type that supports the superset of all the field values resulting from the **Union**.
- The resulting data type from combinations of “**different**” data types (e.g., number with **Date**, **Date** with **Boolean**, ...) **to String**.

If you want to know more about this, you may click “[F.9.3 What are the output fields of a Union?](#)”.

For example, the next Query<sup>102</sup>:

```
SELECT "Madrid" AS Cap_City, "No_comments" AS Comments, #1/1/2018# AS fecha
FROM T_Numbers WHERE Num=1
UNION
SELECT Capital, DateSerial(Cal_Year, 1, 1), District
FROM T_Capital_Rainfall_District
```

does not produce an error, and rather, it returns a record-list converting all the values in the second and third fields to *String* data type. Even though this Query “works”, the record-list it returns is very likely **not** what was intended by the programmer.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

#### J.16.4 What data type bugs can I get with aggregate functions?

Most SQL **aggregate function** and **domain aggregate functions** have restrictions on the data/field types of its argument. If you apply an aggregate function over an unsupported data type the Query will **crash** and MS-Access will most likely show one of the following error messages:

- “Data type mismatch in criteria expression.”
- “Cannot have Memo, OLE, or Hyperlink Object fields in aggregate argument (Value\_expr).”  
where “Value\_expr” is the expression argument of the aggregate function.

You **fix** this by correcting the argument of the aggregate function so it contains only the supported data types. If you want to know about what are the supported data types in aggregate functions, click “[F.7.18.7 What is a summary and grouping of aggregate functions?](#)”.

#### J.16.5 What data type bugs can I get with VBA functions?

When a VBA function is invoked, the system checks that the data type of **each** actual function argument is **the same** (or a compatible one) as the data type **declared** for that argument in the function definition. If there is a data type mismatch, the function will crash. Therefore, declaring the specific data type of each argument when you write your user-defined functions is **very useful** to prevent possible errors and mistakes done in function invocation.

However, if you want that a given user-defined function argument may take **Null** as a value, declaring a specific data type for that argument in the function definition will cause the function to **crash**, because **Null does not belong to any specific VBA data type**. Therefore, whenever you expect that a function argument may be **Null** (which is **very** frequent), you should declare the argument data type as *Variant*, and you start the function code by checking if any of the arguments is **Null**. Depending on the existence of **Null** arguments, and the specific semantics of the function, you have to decide what the function should do. The most usual result of one or more **Null** argument(s) is that your user-defined function will return **Null**, but this is not always the case.

Regarding built-in functions, remind that MS-Access uses weak typing, and therefore, it

---

<sup>102</sup> This is the Query “J\_Types\_Union” from file “Company\_Database.accdb”.

will try to interpret the argument of the built-in function as belonging to the data type declared for the argument. For example, if you write the expression “**Log("23")**”, where “**23**” is a text string enclosed in double quotes, this will work, because MS-Access will convert the text string “**23**” to the number “**23**”. However, if you write the expression “**Log("a3")**”, this will **crash**, because you are providing a text string as an argument of a function that expects a number.

A very small set of functions can return values of **more than one** data type, depending on the values of its arguments. An example of such functions are “**Choose()**” and “**Switch()**”, that depending on the arguments may return a **numeric-like** or a *String* data type.

## PART K. USEFUL DESIGN ADVICE

If you want advice on **database design**, you may click:

- “[K.1 What are good practices in my Table design?](#)”
- “[K.2 What are other good practices in my database design?](#)”
- “[K.3 How do I structure and optimize a distributed database?](#)”

If you want advice on **Query design**, you may click:

- “[K.4 What Query design principles should I follow?](#)”
- “[K.5 Why and how should I carefully handle Nulls in my Queries?](#)”
- “[K.6 What are some useful models of SQL code?](#)”
- “[K.7 Why and how do I design a fast database and fast Queries?](#)”
- “[K.8 Why should I avoid using Decimal data types?](#)”

If you want advice on **VBA function design**, you may click:

- “[K.9 How do I write my user-defined VBA functions and database Subroutines?](#)”

If you want help to find the specific section you want, among the ones covering the same concept, you may click:

- “[K.10 What elements/concepts are explained in various places?](#)”

### **K.1 What are good practices in my Table design?**

You may click:

- “[K.1.1 Why should I write field descriptions in my Table fields?](#)”
- “[K.1.2 Why should I add validation rules to my Tables?](#)”
- “[K.1.3 Why should I prevent Nulls in my Table fields?](#)”
- “[K.1.4 Why should I add a “Comments” field to my Tables?](#)”
- “[K.1.5 Why should I add at least one Date/Time field to my Tables?](#)”
- “[K.1.6 How to prevent errors in Short Text fields?](#)”
- “[K.1.7 Why should I configure drop-down menus to enter data?](#)”
- “[K.1.8 What are good practices in configuring my drop-down menus?](#)”
- “[K.1.9 How do I configure a drop-down menu in a Date/Time field?](#)”

#### **K.1.1 Why should I write field descriptions in my Table fields?**

Because field interpretation is usually ambiguous. If you have a Table of invoices, with a field called “Invoice\_Date”, you may think it is pretty obvious what it means. However, there may be ambiguity between the date the invoice was internally **approved**, the date the invoice was **accounted**, the date the invoice was **sent** to the customer, the date the invoice was **received** by the customer, the date the invoice was **paid** and so on.

It is therefore **very good practice** that you use the field property “**Description**” to explain with some detail what is the exact and precise meaning of each field in your database Tables. Notice that MS-Access will **show** the content of the field description in its bottom window frame each time that you select the **field’s value** (click [B.5.3](#)): this automatically provides a useful information about the specific meaning of the field.

If you want to know how to add a field description to your fields, you may click “[D.5.1.1 What is the “Description” Table field property?](#)”.

### **K.1.2 Why should I add validation rules to my Tables?**

Because you may want to **restrict** the values in your Tables **and** because you want to **check** the entered values **for errors**.

Regarding **checking errors** when entering data, it is **extremely frequent** to introduce **erroneous** data in Tables. Table errors usually come from a misunderstanding of the field meaning, typing-in mistakes, problems with cut/paste buffer or data representation mismatch when importing data from other applications. If you want to know more about cut/paste problems, you may click “[L.4.4 How do I fix errors when pasting records into a Table/Form?](#)”.

If you want to know more about why and how to **restrict** the values entered in your Tables, you may click:

- “[K.1.2.1 Why should I add field validation rules to my Table fields?](#)”
- “[K.1.2.2 Why should I add record validation rules to my Tables?](#)”
- “[K.2.10 How do I restrict the values introduced in my Table fields?](#)”.

Wrong Table data causes big problems. If you are lucky, the erroneous Table data will cause a Query crash, exception-values or some other very clear manifestation of the error. If you are **unlucky**, **nothing will seem wrong**, but you will be getting wrong results from your database. Based on your wrong database results you will be making **wrong decisions**, until the mistake is detected at some point in the future.

#### **K.1.2.1 Why should I add field validation rules to my Table fields?**

Because the **field validation rule** will prevent a number of errors in the data entered in the field. If you want to know how to set a field validation rule, you may click “[D.5.1.5 What is the field “Validation Rule” Table field property?](#)”.

Notice that a **field** validation rule **cannot involve any other field in the record**, and therefore you can only establish a condition over **each individual field value**. This is clearly a limitation, but if you want to establish conditions involving several fields, you can write a **record validation rule** (click [K.1.2.2](#)).

I advise you (almost) always configure a field validation rule. This is particularly important in Short Text fields because invisible characters pose a big risk of errors in these fields. If you want to know more about these problems, you may click “[L.7 How do I fix errors with Short Text or String fields?](#)”.

I am now listing a few frequent field validation rules that you may find useful:



Rule that prevents any space, tab “Chr(9)” or line-feed “Chr(10)” within a text field:

```
0=InStr([Fld_name];" ") + InStr([Fld_name];Chr(9)) + InStr([Fld_name];Chr(10))
```

Rule that prevents any leading or trailing spaces, any in-string line-feed “Chr(10)” and any in-string tab “Chr(9)” within a text field. Notice this rule does **allow** spaces **within** the text field:

```
([Fld_name] = Trim([Fld_name]))
AND (0 = InStr([Fld_name];Chr(9)) + InStr([Fld_name];Chr(10)))
```

Rule enforcing that a **Date/Time** field has **zero-date** in **all** its values. This is, that the date in all the values is December 30th 1899 (integer part equal zero), as if it had no date information.

```
Fix([Fld_name]) = 0
```

Rule enforcing that a **Date/Time** field has **zero-time** in **all** its values. This is, that the time in all the values is 0:00:00 (fractional part equal zero) as if it had no time information:

```
Fix([Fld_name]) = [Fld_name]
```

### **K.1.2.2 Why should I add record validation rules to my Tables?**

Even if all the record’s fields satisfy their own **field validation rule** (click *K.1.2.1*), the record data as a whole may be erroneous. For example, you can have a Table of contracts with fields “Start\_Date” and “End\_Date”. You possibly introduced a field validation rule in each field, stating that the field’s value should be higher than today’s date. This is good practice but would not prevent entering a **wrong** record having a value of “Start\_Date” **after** “End\_Date”.

You can prevent such errors using a **record validation rule**. Record validation rules allow to involve **all** the Table fields, and therefore you can establish validation expressions that involve several fields. If you want to know how to set a record validation rule, you may click “*D.8.1 How do I configure a record validation rule?*”.

My advice is you (almost) always configure a record validation rule.

### **K.1.3 Why should I prevent Nulls in my Table fields?**

Because **Null** causes the following problems:

- A Table field that **contains Null cannot** be a **Key** field, even if it is configured as “**Required=Yes**”.
- A Table field that **can contain Null** (i.e., configured as “**Required=No**”) **cannot** be a **Key** field.
- A Table field that **can contain Null** (i.e., configured as “**Required=No**”) does **not** prevent **duplicate** records, even if it has been configured as “**Indexed=Yes, without duplicates**”, because **several** different records can contain **Null** in the field.

If you want to know more, you may click “*K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?*”.

- A Table field that **contains Null** must be carefully handled in Queries that use the

said field (click K.5).

Preventing **Null** in a Table field is very simple. You only need to configure the field property “**Required=Yes**”. When a field is configured as “**Required=Yes**”, MS-Access will not allow you to introduce any new record in which this field value is empty, i.e., a record with a **Null** in this field.

If you want to configure a field as “**Required=Yes**”, open the Table in “**Design View**” (click B.4.1.3) and click on the corresponding field **row** want in the top sub-pane. In the tab “**General**” (placed at the bottom sub-pane) you will see the property row “**Required**”. Right-click on the rightmost side of the “**Required**” row and click “**Yes**” from the drop-down menu.

Notice that if you **already** had records with **Null** in a field, and you configure the field as “**Required=Yes**”, MS-Access will present you a warning, but the **Nulls will stay in the Table records**. If you want to **remove** the **Nulls**, do a **bulk-change** of your Table data (click E.7).

#### **K.1.4 Why should I add a “Comments” field to my Tables?**

Because on many cases you will want to **complement** the **standardized data** that each record contains with **informal text** in order to **improve the interpretation** of each record and avoid misunderstandings.

Adding to your Tables a **Short Text** field for **comments** you may introduce textual information whenever you need it, to add complementary clarifications to given specific records. For example, in a Table of invoices a few records may have comments like “This invoice is the one we took to court”, “This invoice had a problem in its delivery deadline” or similar useful comments.

If some specific comments become frequent and relevant enough, it is better to add a specific field to the Table. For example, if you use the “**Comments**” field to put in **every** record “This invoice is paid” or “This invoice is not paid”, it would be a better design to add a **Yes/No** field called “Paid” to your Table of invoices. This would identify which ones are paid (and which ones are not) in a standard way that can be processed in a much more robust way than a text field. Notice that even if you add such a “Paid” field, **it still makes sense** to have the “**Comments**” field, where you can add any other useful information that is not properly captured in the record’s fields.

I advise you add **one** field named “**Comments**”, with field type **Short Text** to (**almost**) **all** your Tables. Exceptions may be Tables with absolutely straightforward information (e.g., a Table with calendar years) and Tables that already have one (or more) **Short Text** field(s) that are suitable to introduce a textual description that can clarify the meaning of the record’s data. Following my general advice over **Short Text** fields, my advice is that you configure this field as “**Required=Yes**”, as “**Allow Zero Length=No**” and having a default **visible** value (e.g., “**Default Value=-**”).

#### **K.1.5 Why should I add at least one Date/Time field to my Tables?**

Because information **changes** along time and because you will want to extract records **based on time information**.

To clarify the **first** reason (i.e., information **changes** along time), imagine you have a Table called “T\_VAT\_Percentages” that records the different VAT percentage values that may be applied to each of the invoices in your Table of invoices. This VAT Table may look like:

VAT_Class	VAT	Comments
General	21.0%	-
Luxury	33.0%	-
Reduced	11.5%	-
Tobacco	45.0%	-
Gasoline	55.0%	For personal use: truckers are applied the General VAT_Class

This is perfectly fine until there is a regulation change and Tobacco VAT goes up to 60.0% with date of effects 31-Oct-2021. You could think that this is as simple as replacing 45.0% by 60.0% in the corresponding record. However, this is **wrong** because **existing** Tobacco invoices should be applied a 45.0% VAT, while **forthcoming** Tobacco invoices should be applied the 60.0% VAT. It is therefore much better having designed your Table “T\_VAT\_Percentages”, from the very beginning, with a “First\_Day” and “Last\_Day” **Date/Time** fields. The Table would then look like:

VAT_Class	VAT	First_Day	Last_Day	Comments
General	21.0%	1-Jan-2018	31-Dec-3000	-
Luxury	33.0%	1-Jan-2018	31-Dec-3000	-
Reduced	11.5%	1-Jan-2018	31-Dec-3000	-
Tobacco	45.0%	1-Jan-2018	31-Oct-2021	Old VAT value.
Tobacco	60.0%	1-Nov-2021	31-Dec-3000	-
Gasoline	55.0%	1-Jan-2018		For personal use: truckers are applied the General VAT_Class

With this VAT Table, you are much safer in respect to changes in VAT regulation. You are even safe if new VAT classes are introduced, or if former classes are removed. Obviously, you have to design your Queries so that each invoice is matched with the correct VAT percentage based on each invoice “VAT\_Class” and date, in respect to the “First\_Day” and “Last\_Day” fields of the Table above.

To clarify the **second** reason (i.e., you will want to extract records **based on time information**), imagine you have a Table “T\_Former\_employees” that looks like:

Name	Fired	Email	Comments
Doe, John	No	Johny345@gmail.com	-

Name	Fired	Email	Comments
García, Juan	No	Juan.garcia@ABC.com	-
Du Pont, Pierre	No	Pierre.du.pont@red.fr	-
Li, Xi	No	Xi.Li.Alice@big.cn	Retired

You use this Table to invite your former employees to open company events, to send them job advertisements, and similar communications. However, after ten years this Table becomes very large (imagine this is the database of Foxconn that has 803,126 active employees!). Then, you **do not want** to send e-mails to **all** your former employees because they are too many and/or because you do not find useful to address employees that left long ago. It is therefore much better having designed your Table “T\_Former\_employees”, from the very beginning, with a “Last\_day” **Date/Time** field. Based on that field it is trivial to limit the former employees to which you send letters to only the ones that have left less than “x” years ago.

My advice is therefore that (almost) **all** your Tables should have **at least one** meaningful **Date/Time** field.

### **K.1.6 How to prevent errors in Short Text fields?**

**Short Text** fields are more prone to errors than **Number**, **Date/Time**, **Yes/No** or **Currency** fields because of invisible characters, non-English characters, typos and the treatment of strings in some operators (click *G.5*). If you want to know more about problems you can encounter with **Short Text** fields, you may click “*L.7 How do I fix errors with Short Text or String fields?”*”.

It is therefore a good practice to be particularly careful with **Short Text** fields.

My advice is that you follow the next good practices:

- Configure all your **Short Text** fields as “**Required=Yes**”  
This is a general good practice for any field (click *K.1.3*) and is particularly important for **Short Text** fields. It prevents errors arising from **Null** (click *J.14*).
- Configure all your **Short Text** fields as “**Allow Zero Length=No**”  
This prevents errors arising from the **zero-length** string (click *L.7.8*).
- Configure all your **Short Text** fields with some “**Default Value**”  
This prevents errors from having to type-in a specific value for each new record.
- Configure, as much as possible, your **Short Text** fields as a **slave field** in a Relationship with referential integrity  
**Slave** fields can **only** take values from their **master** field in a Relationship with referential integrity. This means that there cannot be mistyped, **Null** or erroneous values in the **slave** field, because each and every value **corresponds** (guaranteed by the database engine) to a value in its **master** field (check *C.11.1*).
- Configure, whenever possible, your **Short Text** fields with a **drop-down menu** with property “**Limit to List=Yes**”  
This is a general good practice for any field and is particularly important for **Short**

**Text** fields. Entering data with a drop-down menu is faster and reduces substantially the possibility of errors. In particular, if the drop-down menu contains all the possible values for the field, configuring its property “**Limit to List=Yes**” prevents the user from mistakenly changing the value. If you want to configure a drop-down menu, you may click “*D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?*”.

- Configure, as much as possible, your **Short Text** fields with a **field validation rule**  
This is a general good practice for any field and is particularly important for **Short Text** fields (click *L.7*). I strongly advise you to configure field validation rules that **prevent** invisible characters in your text strings (click *K.1.2.1*). If you want to configure a **field** validation rule, you may click “*D.5.1.5 What is the field Validation Rule” Table field property?*”.
- Configure, whenever possible, a **record validation rule** involving the **Short Text** fields  
This is a general good practice for any field and is particularly important for **Short Text** fields. If you want to configure a **record** validation rule you may click “*D.8.1 How do I configure a record validation rule?*”.
- Configure, unless you have a powerful reason, **Field Size=255**  
Unless you need to save space for a very good reason, my advice is you always configure the maximum size for all **Short Text** fields.

### **K.1.7 Why should I configure drop-down menus to enter data?**

Because drop-down menus allow much faster and convenient data entry, and also prevent errors.

My advice is you configure as many drop-down menus as you can in your Table fields.

If you want some useful advice on how to do a good configuration of your drop-down menus, you may click:

- “*K.1.8 What are good practices in configuring my drop-down menus?*”

If you want to know a trick to configure drop-down menus on **Date/Time** fields, you may click:

- “*K.1.9 How do I configure a drop-down menu in a Date/Time field?*”

If you just want to know the procedure itself to configure drop-down menus, you may click:

- “*D.11.1 How do I configure a drop-down menu to enter data in a Table field?*”
- “*D.11.4.1 How do I configure a new drop-down menu directly in a Form field?*”

### **K.1.8 What are good practices in configuring my drop-down menus?**

I explain here some **good practices** and **advice** on how to do a **good configuration** of your drop-down menus.

If you rather want **the mechanics** of configuring drop-down menus, you may click:

- “D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?”
- “D.11.4.1 How do I configure a new drop-down menu directly in a Form field?”

To start with, my advice is you always configure “**Display Control=Combo Box**” because it is more flexible and has more configuration options than “**List Box**”. To find additional information, you may click:

- “K.1.8.1 What “Row Source Type” and “Row Source” should I use in my drop-down menus?”
- “K.1.8.2 Should I create a Relationship from an auxiliary Table used in drop-down menus?”
- “K.1.8.3 Should I configure “Limit to List=Yes” in my drop-down menus?”

### **K.1.8.1 What “Row Source Type” and “Row Source” should I use in my drop-down menus?**

Remind that the “**Row Source Type**” property indicates if values in your drop-down menu are taken from: a) the **values** in the **first field** of a Table, Query or SQL operation; b) an **explicit list** of values; c) the **field names** of a Table or Query.

Remind that the “**Row Source**” property indicates **where to get** the values to be shown in the drop-down menu.

#### **In case the drop-down menu is in a slave field**

Then, the drop-down menu values **must** match values from its **master** field, and you should set “**Row Source Type=Table/Query**”. There are three usual cases:

- If the master Table is **small** (e.g., it has 20 records):  
You can use the Table name as your “**Row Source**”.
- If the master Table is **large**, and it is **simple** to identify what values you want to show:  
you can use a **Select** operation over the master Table as your “**Row Source**”. In the **Select** operation you can restrict the values being shown (e.g., to avoid showing old deprecated values) with a “**WHERE**” clause and you can also configure the order of values in the menu using an “**ORDER BY**” clause.
- If the master Table is **large**, and it is **complex** to identify what values you want to show:  
You can use a specifically designed Query as your “**Row Source**”. For example, for a field containing a customer name, you could decide to show in the drop-down menu the twenty customer names that submitted the highest amount purchased in the last month. You can therefore design a specific Query to produce that twenty names.

**In case the drop-down menu is not in a slave field and the menu values will change**

**little or none at all**

Then, you have two usual cases:

- In case the values in the drop-down menu **will not change for sure** and they are not very many (e.g., below 15)  
My advice is you configure “**Row Source Type=Value List**” and use an **explicit value list** as your “**Row Source**”. One example would be entering month names, or quarters: both are values that will not change for sure, and the number of values is not very long.
- In case the values in the drop-down menu **will possibly change** and/or they are very many (e.g., above 15)  
My advice is you configure “**Row Source Type=Table/Query**” and you place the values for the drop-down menu in an **auxiliary Table**. Using an auxiliary Table has the following advantages:
  - It is easier to update the values than when they are an **explicit list** in the “**Row Source**” property.
  - In case the values are used in drop-down menus for several fields in several Tables, you only need to update the values once instead of having to do it in each of the drop-down menus.
  - You may have in the auxiliary Table, in addition to the field with the drop-down menu **values**, an additional field with a descriptive text, an additional field with comments, an additional field to order the values over it, or any other auxiliary fields you may find useful.

**In other cases**

A couple other cases you may find useful are the following:

- You get the values of the menu from the **field itself**.  
For example, if the field is for entering a complaint from a customer, the drop-down menu can show the 20 most frequent complaints over the past month. In this case, “**Row Source Type=Table/Query**” and “**Row Source**” is a Query over the Table field itself.
- You have a **Transform Query** whose field **names** you want to use in a drop-down menu  
Remind that the field **names** of a **Transform Query** correspond to the field **values** of an SQL operation. In this case, “**Row Source Type=Field List**” and “**Row Source**” is the name of the **Transform Query**.

**K.1.8.2 Should I create a Relationship from an auxiliary Table used in drop-down menus?**

In case you want that the fields that use an auxiliary Table for their drop-down menu **only** take values existing in the auxiliary Table, you can achieve this by configuring a Relationship. You create a Relationship (with referential integrity and with the option “**Cascade Update Related Fields**” set) between the field in the auxiliary Table (as the **master** field) and each of the fields (as **slave** fields) that use the auxiliary Table in their corresponding drop-down menu. If you create such a Relationship, it has the following

implications that you need to be aware:

- **Advantage:** it is completely guaranteed that **each** slave field only takes its values from the ones existing in the auxiliary Table.
- If a value in the auxiliary Table is **modified**, its new value will be **automatically updated** in **all** the corresponding records in **all** the slave Tables. This is usually an advantage, but it can also be a disadvantage. You anyway have to be aware of this.
- **Disadvantage:** If a value in the auxiliary Table is **deleted** (because it is obsolete), you will either have to delete the Relationship, or delete all the records that had that value.

An elegant way of avoiding the disadvantage above is by adding a **Date/Time** field named “Date\_Obsolete” (or something similar) to the auxiliary Table. In this way, you can **keep all** the “old” (obsolete) values in the auxiliary Table setting their “Date\_Obsolete” to the corresponding date, and you can **add new** values as needed. For active values you assign to them a “Date\_Obsolete” date that is long in the future (e.g., 31-Dec-3000). Doing this, you may show in the drop-down menu only the values that are **not** obsolete, by adding the following clause to the **Select** operation:

```
WHERE Date_Obsolete >= Date()
```

The obsolete values that the user should not introduce any more are therefore **not** shown in the drop-down menu.

Adding such a **Date/Time** field is considered a good practice (click *K.1.5*).

You can also add Table validation rules, checking that obsolete values are only accepted when some **Date/Time** field in the records is earlier than some given date. This is technically very sound, but the expression in the validation rule may become large and cumbersome (remind you **cannot** use user-defined VBA functions in validation rules).

### **K.1.8.3 Should I configure “Limit to List=Yes” in my drop-down menus?**

Remind that configuring “**Limit to List=Yes**” only enforces that **newly entered** values belong to the ones of the drop-down menu. Therefore, you **can paste** in the field values **not** contained in the drop-down menu and the field can **store already existing** values **not** contained in the drop-down menu. If you want to know more about how to restrict the values in your fields, you may click “*K.2.10 How do I restrict the values introduced in my Table fields?*”.

**If the drop-down menu contains all the acceptable values to be currently entered in the field**

You should then configure “**Limit to List=Yes**”. This prevents the user from mistakenly editing the selected value and entering a wrong value that is not contained in the drop-down menu.

Even if the field where the drop-down menu is configured is a **slave** field in a Relationship with referential integrity, it is still valuable to configure “**Limit to List=Yes**”. The main reason is that the master field contains **all** the possible values that the field with the drop-down menu can contain, but the menu contains the possible values that can be **currently**



entered, which can be a subset of all the possible values. If you want to know more about this subtle (but relevant) difference, you may check my explanation on auxiliary Tables clicking on *K.1.8.1*.

An added advantage of configuring “**Limit to List=Yes**” when the field is a **slave** field is that in case the user mistakenly edits the selected value (to a value non-existing in the **master** field) the user will get the MS-Access error notification right at the moment of having entered the value in this field, and does not have to wait instead until having entered all the values in the record (which is the moment when the MS-Access will check the Relationship values).

**If the drop-down menu does not contain all the acceptable values to be currently entered in the field**

You should then configure “**Limit to List=No**”. This allows the user to choose a value from the drop-down menu, and then edit it to produce the valid value she/he wants to enter. This is particularly useful when the drop-down menu produces values from the **field itself** (check the **first bullet** from paragraph “*In other cases*” from *K.1.8.1*).

### **K.1.9 How do I configure a drop-down menu in a Date/Time field?**

MS-Access **does not** allow configuring a drop-down menu in a **Date/Time** field. It allows you to configure a date-picker, but this is a very different functionality.

If you **really need** a drop-down menu in a **Date/Time** field, there is a **smart trick** to do it. The trick is based on the equivalent internal representation between **Date/Time** values and **Number-Double** values (click *D.4.5*). It consists of configuring the field as a **Number-Double** but displaying its values with **Date/Time** format. In this way, MS-Access allows you to configure a drop-down menu in the field. The configuration of the menu is slightly tricky, but it works. If you want to do this, follow the next steps:

Configure the Table field as “**Field Type=Number**” and “**Field Size=Double**” (click *D.4, D.4.3* and *D.4.3.5*).

Then configure the field’s **Format** property as a **Date/Time** format either **predefined** (click *H.6.1*) or **custom** (click *H.6.2*).

Now you configure the drop-down menu (click *D.11.1.1* for Tables or *D.11.4* for Forms). You have to configure the “**Row Source**” property to produce **two columns** in the drop-down menu. To do this, configure the “**Row Source**” property of this field with the following content:

```
SELECT CDb1 (Menu_Date), Menu_Date FROM Query_Date ;
```

where “**Query\_Date**” is the **name** of the Query that you have written to produce the **Date/Time** values that you want to be shown in your drop-down menu, and “**Menu\_Date**” is the Query **field name** corresponding to the **Date/Time** values to be shown in the menu. If you have not written a specific Query, and you are taking the values from a Table, then write the Table name instead of “**Query\_Date**” in the **Select** operation above.

Then, configure the field property “**Column Counts=2**” (to display both fields) and the field property “**Column Width=0.01";1"**” to **conceal** the first column (if shown, it can

only create confusion) while **showing** the second column. Click *D.11.5.1* to know how to configure these two properties.

**Be aware** that with this field configuration MS-Access **will not allow you to type-in Date/Time** values into this field. Because the field is configured as a **Number-Double**, MS-Access will **only** allow you to type-in numbers. You can of course type-in a **Date/Time** value as its numeric equivalent number, but this is obviously cumbersome. The same applies if you try to paste a value having **selected** one **field's value** (click *B.5.3*): only numbers can be pasted. **However**, if you **select one field** (or a range of record fields, click *B.5.2*) (to see the difference between both selections, click *B.5.4*), **you** can directly paste **Date/Time** values.

Therefore, if you use this trick, you can enter values by using the drop-down menu, by pasting values, and by the cumbersome approach of typing them as a number.

I strongly advice that when you use this field in your expressions, you **always** enclose it in a "**CDate()**" type conversion function, in order to make sure that the values from the Table are correctly interpreted as a **Date/Time** field type instead of as a **Number-Double** field type-size.

Notice that you **cannot** use this same trick to have a drop-down menu in a **Currency** field because the internal storage format of **Currency** and **Number** fields is **different**.

## **K.2 What are other good practices in my database design?**

In addition to the good practices presented in "*D.2 How do I carefully assign good names from the very beginning?*" and "*K.1 What are good practices in my Table design?*", this chapter contains the following useful tips and recommendation on other aspects of your database design:

- "K.2.1 Why should I hide unfrequently used objects?"
- "K.2.2 Why should I add a "T Numbers" Table with integer numbers?"
- "K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?"
- "K.2.4 What are the interactions between Relationships and "PrimaryKey", "Required" and Table indexes?"
- "K.2.5 What are the actual fields and the actual field order in a Table/Query/Form?"
- "K.2.6 What is the difference between a Calculated field and automatically introducing a value using a Form?"
- "K.2.7 How should I back-up MS-Access database files?"
- "K.2.8 How do I store a list of values, instead of a single value, in a Table field?"
- "K.2.9 Why and how should I maximize Table data correctness and coherence?"
- "K.2.10 How do I restrict the values introduced in my Table fields?"

### **K.2.1 Why should I hide unfrequently used objects?**

Because hiding them allows you to find faster the objects (Tables, Forms, Queries) that

you use more frequently.

Some examples of unfrequently used objects are:

- Queries used only by other Queries, but not invoked by the user.
- Temporary Tables.
- Tables that contain quasi-static information (e.g., a list of countries, a list of states, a list of month names, ...).
- VBA modules with user-defined functions

If you want to hide/unhide objects, you may click [B.4.2.4](#), [B.4.2.5](#) and [B.4.2.6](#).

### **K.2.2 Why should I add a “T Numbers” Table with integer numbers?**

Because it will be extremely useful as an auxiliary Table for writing your SQL Queries.

Such a Table only has one field (that I call “Num”), and its records contain a certain range of **integer numbers** (e.g., from **-100** and **100**). You can see an example of such Table in the database file “Company\_Database.accdb” (click [A.1](#) to download it).

Some examples of cases where such a Table is useful are:

- Click “[K.6.1 How do I generate/create record-lists?](#)”
- Click “[K.6.2 How do I replicate record-lists?](#)”.
- Click “[K.6.3 How do I produce totals in addition to individual results?](#)”.
- Click “[K.6.5 How do I convert columns into rows?](#)”.
- Click “[K.6.7 How do I produce the non-matching records of a Join operation?](#)”.
- Click “[K.6.8 How do I write a Full-Outer-Join?](#)”.

Since this Table is **static**, you may consider placing it as a **local** Table in the **frontend** files (click [K.3](#)).

### **K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?**

I have summarized the interactions between **Nulls**, **duplicate values**, **duplicate records**, **indexing** and **Key field(s)** in the following table, that I hope you will find useful. The table presents in each cell the specific result that you get when you configure in one field (or a group of fields) the six possible combinations between the **three indexing types** and whether the field (or **all** the fields in the group) are configured **with Nulls** (i.e.,

Required=No) or without Nulls (i.e., Required=Yes):

Configuration of one field or of one group of fields	No Indexing	Indexing with duplicate values	Indexing without duplicate values
<b>With Nulls</b> One or more fields with <b>Required=No</b>	<u>Independent fields</u> (one or more of them with <b>Nulls</b> )	<u>Index</u> with duplicate values and with <b>Nulls</b>	<u>Index</u> without duplicate values, and with <b>Nulls</b>
<b>Without Nulls</b> <u>All</u> fields in the group with <b>Required=Yes</b>	<u>Independent fields</u> (all of them <u>without</u> <b>Nulls</b> )	<u>Index</u> with duplicate values but <u>without</u> <b>Nulls</b>	<b>Candidate Key =&gt;</b> <u>Index</u> without duplicate values and <u>without</u> <b>Nulls</b> . <b>No duplicate records</b>

Notice that an index **without duplicate values** and **with Nulls** does **not** prevent duplicate records in the Table, because there can be many records having **Null** in some of the index fields and the **different values** in the other fields.

If you configure one (or more) field(s) as the **Primary Key** of the Table, MS-Access will **automatically** configure over the **Primary Key** field(s) an associated index **without duplicate values and without Nulls**. This, it will **automatically** configure each and every **Primary Key** field(s) as “**Required=Yes**” and will also configure an index **without duplicate values** for the **Primary Key** field(s).

As it does for any index **without duplicate values**, MS-Access will check that the Table does not contain any record with duplicate values on the **Primary Key** field(s): in case the Table contains records with duplicate values, MS-Access **will not allow** you to do the configuration of these **Primary Key** fields (click L.2.6).

Also, MS-Access will check that the Table records contain no **Nulls** on the **Primary Key** field(s). Unlike MS-Access does for a “normal” index without duplicate values, for the case of the **Primary Key** fields, MS-Access **does not allow** to configure it in case the Table contains any existing record with **Null** in any of the **Primary Key** fields (click L.2.7).

Once the **Primary Key** field(s) are configured, MS-Access will **not** allow you to change neither the **indexing** nor the “**Required**” configuration of the **Primary Key** field(s).

I want to highlight that it is **very different** to configure an **index** to **ignore Nulls**, than to configure the index **fields** to **prevent Nulls** (i.e., configuring all fields as “**Required=Yes**”). Configuring an **index** to **ignore Nulls** just makes the index slightly more efficient, but it **does not** prevent **Nulls** in the corresponding fields of new Table records. Remind that when I say that an index is **without duplicate values and without Nulls** it means that the index is configured as without duplicates and that all the index fields are configured as **without Nulls** (i.e., “**Required=Yes**”).

As a final remark, for the case of any **Short Text** field, MS-Access does **not** automatically set “**Allow Zero Length=No**”, when configuring it as a **Key** field, nor when setting

“**Required=Yes**” in the field, nor when configuring indexing over the field. At the same time, it is very unlikely that you will need zero-length strings in your database. For these reasons, I strongly advise you to **manually** set “**Allow Zero Length=No**” in every **Short Text** field. Doing this much reduces the risk of making mistakes between a zero-length string, a **Null** and an invisible string.

#### **K.2.4 What are the interactions between Relationships and “PrimaryKey”, “Required” and Table indexes?**

Relationships have a strong interaction with the field properties “**PrimaryKey**” and “**Required**”, and with the Table **indexes without duplicate values** and **without Nulls**.

In Relationships with referential integrity (either **one-to-one** or **one-to-many**) **both fields in each and every master-slave field pair must have the same field type and field size**, and also, the **master field(s)** of the Relationship **must** have an associated index **without duplicate values and without Nulls**. Just to clarify, it is **not** enough if a **subset** or a **superset** of the **master field(s)** have an index **without duplicate values and without Nulls**: it must be exactly the master field(s) having that index. The **master Table** may have **other** indexes, and this does not cause any problem.

Remind from *K.2.3* that having an **index without duplicate values and without Nulls** is the same as being a **candidate Key**.

In a **one-to-many Relationship** every **pair** of related fields is of the same field type and field size, the **master field(s)** is/are a **candidate Key** of the **master Table**, and neither the **slave field(s)**, nor **any subset** of them, are a **candidate Key** of the **slave Table**.

In a **one-to-many Relationship** every **pair** of related fields is of the same field type and field size, the **master field(s)** is/are a **candidate Key** of the **master Table**, and **either the slave field(s), or any subset** of them, are a **candidate Key** of the **slave Table**.

The **master** and/or **slave Tables** may have **other** indexes, as long as they satisfy the definitions I just gave.

As you may see, if you want a **one-to-many** or a **one-to-one Relationship**, you need to configure the Table indexes and the “**Required**” property of the **master** and **slave** fields according to the definitions above. Notice that if you configure the **master** fields or any subset of the **slave** fields as a **Primary Key** field, they will automatically have an associated **index without duplicate values and without Nulls**.

Finally, I want to point out that the **slave** fields in a Relationship with referential integrity **cannot** contain any **Null**, even if they have **not** been configured as “**Required=Yes**”. The reason is that the **slave** fields can only take values from the **master** fields, and the master fields cannot contain **Null**, because they must have an associated **index without duplicate values and without Nulls**.

#### **K.2.5 What are the actual fields and the actual field order in a Table/Query/Form?**

If you do a **cut/paste** operation, the actual fields and the actual field order in a Table/Query/Form are the ones you are **actually seeing** in “**Datasheet View**”. Therefore, for cut/paste operations, the Table/Form fields and field order in “**Design View**” and the

Query fields and field order in “**Design View**” or “**SQL View**” are irrelevant.

If you use a Table/Query in a **Union** operation or an **Insert** operation, the **Table** fields and field order are the ones in “**Design View**”, while the **Query** fields and field order are the ones in “**SQL View**”.

### **K.2.6 What is the difference between a Calculated field and automatically introducing a value using a Form?**

Values in **Calculated** fields (click *D.4.8*) are **not stored** in the Table. They are **calculated** from the **other values** in each Table record using the expression configured in the Table field. Each time you do a **Select** operation over this Table involving the **Calculated** field, its values are calculated by the system. Therefore, if you **update** the expression of a **Calculated** field, this will affect **all the values** of this field in **all the already existing** records in the Table.

Values **automatically** introduced by a Form (click *D.10.4.3*) **are stored** in the Table in the corresponding non-Calculated field. Therefore, if you **update** the expression used by the Form to calculate the automatically introduced value, this will **only affect newly introduced records**, and the values of this field in **all the already existing** records in the Table **stay unaltered**.

For example, imagine you have a Table of invoices, and you want to automatically show the invoice amount including VAT (field “**With\_VAT**”), calculated from the invoice amount without VAT (field “**Without\_VAT**”). Imagine that VAT percentage is 21%. You do this by using the expression:

```
With_VAT = 1.21 * Without_VAT
```

either in a **Calculated** field or in a Form that automatically introduces the value in the field.

If the VAT percentage changes at some moment to 20%, and you replace “1.21” by “1.20” in the expression above, the result in the field data is very different. With a **Calculated** field **all** the already **existing** invoices will now show a **wrong** value of field “**With\_VAT**”, while in the case of a Form, all the already existing invoices will have the correct value.

On my experience, it is much more frequent to prefer that all stored values stay the same than preferring that they are all recalculated. For this reason, I think that it is much more frequent to use a Form that automatically introduces the value in the field than using a **Calculated** field. Also, if you are using a Form and in some case you actually want to recalculate all previous values, this is not so difficult to achieve by doing a bulk-change of your data (click *E.7*).

Another difference between a **Calculated** field and a value automatically introduced by a Form is that the **Calculated** field **does not require any storage space**, because it is not stored.

If you are using an **automatically introduced** value by a Form, and you **edit an existing record**, the automatically introduced field **will most likely be recalculated!** You can make sure it is recalculated correctly if you designed the VBA Subroutine that calculates the value to take into account the **date** of the record, assuming that it does have a suitable

**Date/Time** field (click *K.1.5*). You can also just *be careful* and remind that when editing old records, you may have to manually adjust the recalculated values (but notice that “*being careful*” is **very risky**...).

My overall advice is you always use a Form that automatically introduces the value in the field instead of a **Calculated** field, unless you **really must save storage** space. If you want to know how to automatically introduce a value in a field using a Form, you may click “*D.10.4.3 How does my VBA code modify field values in the Form record being edited?*”.

### **K.2.7 How should I back-up MS-Access database files?**

I do not ask if you should do back-up because the answer is always “**OF COURSE!**”: it is a fact that disks fail, computers fail, viruses delete or encrypt files, water leakages damage computers and so on.

It is therefore **absolutely mandatory** to do **daily automatic** backup of your MS-Access database files. And it is **not enough** to just do one automatic daily copy of your latest database files: you should keep the most recent “*n*” **daily copies** of each file, where “*n*” should be **preferably 30** and **never less than 7**.

In this way, if anyone deletes some relevant information, the missing data may be recovered from the backup file of a date **before** the deletion. In case corrupt data is entered, you can recover a correct database file from a date before the corrupt data was entered. Changes occurred from that date have to be manually re-entered taking them from the current corrupted database. This is quite inconvenient, but it is **infinitely worse** the alternative of having lost **all your database** file or having wrong data that you are not very sure which is it.

### **K.2.8 How do I store a list of values, instead of a single value, in a Table field?**

Storing a list of values in a field is considered a **bad practice** and my advice is that you **avoid** using this feature. However, if in spite of my advice you want to do it, I explain it in this section.

MS-Access **allows** you to store a **list of values** in **one field**, instead of just one single value. You can only do it for **Short Text**, **Number** and **Large Number** field types. MS-Access will **not** allow you to use a field with a list of values as part of any index, and consequently, cannot be a simple **Key**, nor part of a composite **Key**. Another restriction is that fields that store multiple values cannot be used in “**WHERE**” or “**GROUP BY**” expressions. They cannot be used in expressions, not even with the “**IN**” operator, that is specific for lists of values.

To store a **list of values** in **one field** configure a **drop-down** menu (click *D.11.1*) in the field and then configure “**Allow Multiple Values=Yes**” (click *D.11.5.1*). You may then **remove** the drop-down menu if you want, but the configuration in the field to store **multiple values** will **remain**. Actually, it is **not possible** to undo the field configuration of storing multiple values and go back to storing a single value. If you want to go back to storing a single value, you have to **remove** the field that you configured to store multiple values and create a **new** field with the usual single-value configuration.

### **K.2.9 Why and how should I maximize Table data correctness and coherence?**

Because **wrong database data** leads to wrong Query results. Based on your wrong database data and wrong Query results you will be making **wrong management decisions**. Data correctness and coherence is therefore paramount in a database.

Once a Query has been working properly for a while, if the Query crashes or the Query produces wrong results, the **most frequent** cause is **wrong data** in your Tables. If you are **lucky**, the erroneous Table data will cause a Query crash, exception-values or some other **clear** manifestation of the error. If you are **unlucky**, **nothing will seem wrong**, but you will be getting wrong results from your database. If you want to know how to fix an erroneous result or a crash in a test-and-proven Query, you may click "[\*J.1 How do I fix an error/crash in a test-and-proven Query?\*](#)".

As you may see, it is **essential** to maximize the correctness of data within each field, within each Table record, among the records of each Table and between the records of different Tables.

You can maximize data correctness and coherence enforcing the following good practices:

- **Restrict the values** entered in your Table fields/record.  
Checking each value entered and restricting it to the set of possible correct values is an excellent practice that can prevent **many** errors.

If you want to know more about how to restrict entered data, you may click "[\*K.2.10 How do I restrict the values introduced in my Table fields?\*](#)".

- Design a **data check Query** and run it **frequently**.  
Validation rules cannot contain user-defined VBA functions. This implies severe limitations on the complexity of error checks that you can do with them. However, a data check Query can use expressions considerably more complex than validation rules and in particular, they can contain user-defined VBA functions. Moreover, a data check Query can perform error checks **among** the records of **each** Table and **between** records of **different** Tables, while validation rules **cannot** do that.

If you want to know more about a data check Query, you may click "[\*K.6.11 How do I design a data check Query?\*](#)".

- Incorporating **data check SQL code to your Queries**.  
In this way, each time you run a Query the coherency of the data it uses is checked, and if an error is detected it is reported. The problem of doing this is that it slows down your normal Queries and makes their SQL code more complex and difficult to debug. The advantage is that you detect data errors earlier. On my view, incorporating data check SQL code to a Query only makes sense for very few Queries that provides **really critical management data**, and you want to minimize the risk of them providing wrong information.

### **K.2.10 How do I restrict the values introduced in my Table fields?**

This section also answers the question:

- **What is the difference between setting a master field, setting "Limit to**



### List=Yes” and setting a validation rule?

You can **restrict** the (new) **values** that a given **Table field** (referred to “the field” and “the Table” along this section) can contain by **configuring** the following database properties/objects:

- **The “Field Type” and “Field Size” of the field**
- **A Relationship (with referential integrity) where the field is a slave field**
- **A simple index without duplicate values over the field**
- **A composite index without duplicate values that includes the field**
- **The field as the Key field**
- **The field as one of the Key fields**
- **The field as “Required=Yes”**
- **The field validation rule**
- **The field within the record validation rule**
- **The field with a drop-down menu with “Limit to List=Yes”**

I now explain each of these cases in more detail.

#### The “Field Type” and “Field Size” of the field

Configuring this **guarantees** that **all the values stored** in the field conform to what can be represented by the chosen “**Field Type**” and “**Field Size**”. If you try to **save** a value, in the field, that does not conform to the field **type** and **size**, MS-Access will **not allow you** to do it (click *L.4.2.1*).

If you want to know more about this, you may click:

- *“D.4 How do I configure a Table field data type and size?”*
- *“G.2.1 What VBA data types vs. Table field types-sizes are equivalent?”*

If **you change** the “**Field Type**” and/or “**Field Size**” properties of the field, you have two cases. If the former field type-size and the new ones are **compatible**, MS-Access will apply the change silently. However, if they are **not compatible**, MS-Access will show a warning message. If you proceed to do the change, MS-Access will perform data conversion **over all the field values in the Table**. If you want to know more about this data conversion, you may click:

- *“I.4.4.1 What are the side effects of changing the “Field Type” and/or “Field Size” properties of a Table field?”*

#### A Relationship (with referential integrity) where the field is a slave field

Configuring this **guarantees** that **all the values stored** in the field **must** exist in the **master** field of one of the existing records of the **master** Table. If you try to **enter** or **modify** a record with a resulting value in the field that does not exist in the **master** field, MS-Access will **not allow you** to do it (click *L.4.3.3*).

If you try to establish a Relationship with referential integrity where the field is a **slave**

field, and it contains values non-existing in its **master** field, MS-Access will **not allow you** to do it (click [L.3.4](#)).

By defining a suitable **master Table** with suitable values in the **master** field, you can therefore restrict the values of the field.

If you want to know more about Relationships, you may click:

- “[C.11 What is a Relationship?](#)”
- “[D.9 How do I create and configure my Table Relationships?](#)”

#### **A simple index without duplicate values over the field**

Configuring this **guarantees** that the field **does not have** a duplicate **value** in two (or more) records. If you try to **enter** or **modify** a record with a resulting value in the field that results in a duplicate value, MS-Access will **not allow you** to do it (click [L.4.3.2](#)).

If you try to configure an **index without duplicate values** over the field, and it **already contains duplicate values**, MS-Access will **not allow you** to do it (click [L.2.6](#)).

If you want to know more about indexes, you may click:

- “[C.8 What is indexing?](#)”
- “[D.7.1 How do I add simple indexes to a Table?](#)”

#### **A composite index without duplicate values that includes the field**

Configuring this **guarantees** that two (or more) records in the Table **do not have** a duplicate **value array** in the index fields. If you try to **enter** or **modify** a record with a resulting value in the field that results in a duplicate value array over the index fields, MS-Access will **not allow you** to do it (click [L.4.3.2](#)).

If you try to configure an **index without duplicate values** over a group of fields that already **contain duplicate value arrays**, MS-Access will **not allow you** to do it (click [L.2.6](#)).

If you want to know more about indexes, you may click:

- “[C.8 What is indexing?](#)”
- “[D.7.2 How do I add composite \(and simple\) indexes to a Table?](#)”

#### **The field as the Key field**

Configuring this **guarantees** that the field has no **Nulls** and **no duplicate values**. If you try to **save Null**, in the field, MS-Access will **not allow you** to do it (click [L.4.2.2](#)). If you try to **enter** or **modify** a record with a resulting value in the field that results in a duplicate value array over the index fields, MS-Access will **not allow you** to do it (click [L.4.3.2](#)).

If you try to configure the field as the **Key field** when it contains **Nulls** and/or **duplicate values**, MS-Access **will not allow you** to do it (click [L.2.6](#)).

Notice that configuring the field as the **Key field** is **not** the same as configuring the field as “**Required=Yes**” (click [D.5.1.7](#)) plus configuring an **index without duplicate values** over the field. The difference is that configuring the field as the **Key field** makes it **impossible** to have **Nulls**, while configuring “**Required=Yes**” **allows Nulls** that were **already** in the Table, and only prevents the **future** saving of **Nulls**.

If you want to know more about a **simple Key field**, you may click:

- “C.10 What are the Table Key(s) and how should I handle them?”
- “D.6.1 How do I configure a Primary Key with only one field?”

### **The field as one of the Key fields**

Configuring this **guarantees** that the field has no **Nulls** and that there are **no duplicate value arrays** over the index fields (that include the field). If you try to **save Null** in the field, MS-Access will **not allow you** to do it (click L.4.2.2). If you try to **enter** or **modify** a record with a resulting value in the field that results in a duplicate value array over the **Key fields**, MS-Access will **not allow you** to do it (click L.4.3.2)

If you try to configure a set of fields including the field as the **Key fields** when they contain any **Null** and/or **duplicate value arrays**, MS-Access **will not allow you** to do it (click L.2.6).

Notice that configuring the field as one of the **Key fields** is **not** the same as configuring the **same fields** as “**Required=Yes**” (click D.5.1.7) plus configuring an **index without duplicate** values over the **same fields**. The difference is that configuring the **same fields** as the **Key fields** makes it **impossible** to have **Nulls** in them, while configuring them as “**Required=Yes**” **allows Nulls** that were **already** in the Table, and only prevents the **future** saving of **Nulls**.

If you want to know more about the **composite Key fields**, you may click:

- “C.10 What are the Table Key(s) and how should I handle them?”
- “D.6.2 How do I configure a Primary Key with several fields?”

### **The field as “Required=Yes”**

Configuring this **enforces** that **newly edited** values in the field cannot be **Null**. However, **records already existing** in the Table **before** you configured “**Required=Yes**” **may contain Null** in the field.

If you try to **save Null** in the field, MS-Access will **not allow you** to do it (click L.4.2.2). Notice however that you **can edit other fields** of an **existing** record containing **Null** in the field, and the resulting edited record with **Null** in the field will remain in the Table.

If you want to know more about “**Required=Yes**”, you may click:

- “D.5.1.7 What is the “Required” Table field property?”.

### **The field validation rule**

Configuring this **enforces** that a **newly edited** value into the field **never violates the field validation rule** (i.e., it does not return *False*). However, records **already existing** in the Table **before** you configured the field **validation rule may have values** in the field that violate the field **validation rule**.

If you try to **save** a value in the field that violates the field **validation rule**, MS-Access will **not allow you** to do it (click L.4.2.6). Notice however that you **can edit other fields** of an **existing** record containing a value in the field that violates the field **validation rule**, and the resulting edited record with a value in the field that violates the field **validation rule** will remain in the Table.

The value restrictions that you can enforce with a field validation rule are somehow limited because its *Boolean* expression **cannot contain user-defined VBA functions**.

If you want to know more about the field validation rule, you may click:

- “D.5.1.5 What is the field “Validation Rule” Table field property?”
- “D.5.1.6 What is the field “Validation Text” Table field property?”

### **The field within the record validation rule**

Configuring this **enforces** that any **newly entered** or **modified** record does not have a value in the field that violates<sup>103</sup> the **record validation rule** (i.e., its *Boolean* expression does not return *False*). However, records **already existing** in the Table **before** you configured the **record validation rule** **may have values** in the field that violate the **record validation rule**.

If you try to **enter** a record with a value, or to **edit** a value, in the field that violates the **record validation rule**, MS-Access will **not allow you** to do it (click L.4.3.I). Notice however that you **can edit other fields** (not involved in the field validation rule) of an **existing** record containing a value in the field that that violates the **record validation rule**, and the resulting edited record with a value in the field that that violates the **record validation rule** will remain in the Table.

The value restrictions that you can enforce with a record validation rule are somehow limited because its *Boolean* expression **cannot contain user-defined VBA functions**.

If you want to know more about the record validation rule, you may click:

- “D.8.1 How do I configure a record validation rule?”
- “D.8.2 How do I configure the record validation text?”

### **The field with a drop-down menu with “Limit to List=Yes”**

Configuring this **enforces** that a **newly typed-in** value in the field belongs to the set of values in the drop-down menu. However, notice that:

- Records **already existing** in the Table **before** you configured “**Limit to List=Yes**” **may have values** in the field **not** belonging to the set of values in the drop-down menu.
- **Pasted** values **over** the field are **not** required to belong to the set of values in the drop-down menu.
- **New pasted** (click E.5.2.3) records and **new inserted** (click F.13.2) records, **may have values** in the field **not** belonging to the set of values in the drop-down menu.

If you try to **type-in** a value in the field that does not belong to the set of values in the drop-down menu, MS-Access will **not allow you** to do it (click L.4.2.5). Notice however that you **can edit other fields** of an **existing** record containing a value in the field that does not belong to the set of values in the drop-down menu, and the resulting edited record with a value in the field that does not belong to the set of values in the drop-down menu will remain in the Table.

---

<sup>103</sup> When I say that a field value violates the record validation rule I mean that it violates it **jointly** with **other** field values in **the same** record.

If you configure a drop-down menu with “**Limit to List=Yes**” over the field, and the Table contains records with values in the field not existing in the drop-down menu, MS-Access **will allow you** to do it without even showing a warning message, and all the non-conformant values will stay in the Table.

If you want to know more about how to properly configure a **drop-down menu**, you may click:

- “*D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?”*”
- “*K.1.8 What are good practices in configuring my drop-down menus?”*”

### **K.3 How do I structure and optimize a distributed database?**

You may click:

- “*K.3.1 How do I organize my database files?”*”
- “*K.3.2 How do I create user-specific views of my shared database?”*”
- “*K.3.3 What are frontend files, backend files and source files?”*”
- “*K.3.4 What are local Tables, linked Tables and source Tables?”*”
- “*K.3.5 How do I get a split database?”*”
- “*K.3.6 How do I create a linked Table?”*”
- “*K.3.7 How do I refresh a Table link?”*”
- “*K.3.8 How do I convert a linked Table to a local Table?”*”
- “*K.3.9 How do I view a Table link?”*”
- “*K.3.10 How do I view and manage Table links?”*”
- “*K.3.11 How do I delete a linked Table?”*”
- “*K.3.12 How do I manage Relationships with linked Tables?”*”
- “*K.3.13 Can I change the file path or name of a source file?”*”
- “*K.3.14 What are the options to lock records in a shared database?”*”
- “*K.3.15 What is the delay of network access to a database?”*”
- “*K.3.16 Why should I make temporal Tables local?”*”

#### **K.3.1 How do I organize my database files?**

The first step is to configure your database for concurrent access. If you want to do this, you may click “*D.13 How do I share a database, having multiple concurrent users?”*”.

The way to organize your database **files** depends mainly on the number of **concurrent** users, on the **database size** and on the **network capacity**, as follows.

##### **One user**

If the database is just for yourself, or for some people that use the same local computer,

you can just have a **single MS-Access file** (extension “.accdb”) in the computer’s **local** drive and use it like you would do with any local file (e.g., an Excel file). This will work well regardless of the database file size. Since this is local access, network capacity does not apply to this case.

The advantage of this approach is that it is the simplest one, and you get the best performance.

You should of course do automatic daily back-up of your database file.

### **Few users and small database and high-speed network**

When there are few users (i.e., two or three), the database file is small (i.e., less than 10 MiBytes) and you have a high-speed network (i.e., a 100 Mbps local Ethernet or better), you can place the MS-Access file in a windows network drive, and access it as you do with a local file. MS-Access incorporates options to support concurrent users (click *D.13*), so this will work well even if two or three users are simultaneously accessing the database.

The way this works is that every time you use the database, your local computer opens the MS-Access file through the network like a regular file, which means transferring a large part of the file through the network.

The advantage of doing this is that you get concurrent access though the network in a very simple way. The disadvantages are that response time is not very good because you have to move through the network a lot of information (click *K.3.15*), and concurrent access works but is not very robust.

You should of course do automatic daily back-up of your database file.

### **Many users or large database or low-speed network**

If you have many users (say, four or more) or the database file is large (say, more than 10 MiB or the network connection has **high transit delay** (click *K.3.15*) and/or **low throughput** (e.g., if you access through mobile data, or through a wide area network), my advice is that you **split** (click *K.3.5*) your database in onw **frontend file** (having several copies) and one **backend file** (both being MS-Access files, click *K.3.3*).

The **backend file** contains the Tables (this is, contains the actual data stored in the Tables).

The **frontend file** contains the Queries, Forms, Reports, VBA code and **links to the Tables** (click *K.3.4*) **stored** in the **backend file**. Each user has a **copy** of the **frontend file** and can run the Queries from it, and also can access the Table’s data stored in the **backend file** (placed in a network drive).

The way this works is that MS-Access implements remote access, so when a user **runs** a Query in her/his **frontend file**, the **Query is executed in his/her local computer**, and MS-Access **brings** the required data (and **only** the required data) from the linked Tables stored in the **backend file**.

The advantage of doing this is that you get **very robust** concurrent access for **many users** and/or for a **large database** and/or accessing through a **low capacity network**. The disadvantage is that the database handling is now more complex, because now each user has his/her own local **frontend** file. If you do a change in the Queries, Forms, Reports or

VBA modules, you have to distribute a **new frontend file** to all the users.

You should of course do automatic daily back-up of your **backend file**. The **frontend files** are far less critical because they are seldom changed (only when you change the Queries and/or the VBA code), and you already have several copies of them. You should anyway also backup it.

### General case

In the general case, you **split** (click *K.3.5*) your database in several **frontend files**, several **backend files** and/or several other **source files** (click *K.3.3*) using **linked Tables** (click *K.3.4*).

**Each** user may have **several local frontend files** with **linked Tables** to **several backend files** and/or other **source files** usually placed on **one or more network drives**. **Each frontend file** contains a **given set** of Queries and VBA code, and has a **number of links** to a number of Tables from **backend files** and/or other **source files**. **Each** user may have **different local frontend files** (depending on the type of user). MS-Access supports linking Tables to **other file formats different** to MS-Access, like linking Tables to Excel, text or html **source files**, so you can have **source files** on different formats. You can **also** have some **local Tables** (in particular temporal Tables for Query processing) in **frontend files**, in addition to **linked Tables**.

The way this works is the same as in the previous case: every time a user **runs** a Query (from one of its **frontend files**), the Query **runs** in her/his **local** computer, and MS-Access brings the required data (and **only** the required data) from the **linked Tables** stored in the corresponding **backend files** and/or other **source files** in the network drives.

This approach gives a **lot of flexibility** to how to organize the information but is obviously **much more complex to manage**. This approach supports many different users, with different needs, considerably large databases and network access with different characteristics in a very flexibly way.

You should of course do automatic daily back-up of all of your **backend files** and/or other **source files**. The **frontend files** are far less critical because they are seldom changed (only when you change the Queries and/or the VBA code), and you already have several copies of them. You should anyway also do backup of them.

### K.3.2 How do I create user-specific views of my shared database?

By producing **specific frontend files** (click *K.3.3*) tailored to the needs of each class of users.

Because each user has her/his own **frontend file**, he/she can have a **different view** of the database. You, as database designer, may for example remove the objects (Queries, Forms, Reports, Modules and Table links) that a given user does not need to be aware of from his/her **frontend file**. This simplifies the user-interface of each user, and prevents accidental errors.

In this way, you can create **different views** of the database, by tailoring each **frontend file** to the needs of each class of users. You can have a user only accessing personnel Tables and having only personnel related Queries. Another user only accessing economical Tables and having only economical related Queries, and so on.

A quite frequent case is that different users see the network drive where the **backend file(s)** are placed with a different unit letter. One user may see a network drive in unit “**D:**” while another user sees **the same** network drive in unit “**F:**”. Therefore, if you produce just **one frontend file** it will not work for all of users. However, if you produce a **specific frontend file** for each group of users that will see the network drive in the same unit letter, this will work fine. Another solution for this problem is to use shortcuts or links, as described in “*K.3.13 Can I change the file path or name of a source file?*”.

### **K.3.3 What are frontend files, backend files and source files?**

I now present the main characteristics of **frontend files**, **backend files** and **source files**:

#### **Frontend files**

- Contain **Queries, Forms, Reports, VBA modules**, some **local Tables** (typically temporal), and **linked Tables** to one (or more) **backend files** and/or one (or more) **source files**.
- They have MS-Access format with “.**accdb**” extension.
- They are usually stored in the **local drive** of **each** user.
- They are usually used by **one** user, that accesses through them, and through the networks, to his/her **backend files** and/or other **source file(s)**.
- Each user usually has one (or more) frontend file(s) used by him/her to access the database services.

#### **Backend files**

- Contain **local Tables** and Relationships **with** referential integrity (and may also contain other database elements)
- They have MS-Access format with “.**accdb**” extension.
- They are usually stored in a **network drive**.
- They are usually used **simultaneously** by **many** users, that access their Tables through their **frontend file(s)** and through the network.
- Each user usually accesses simultaneously to one (or more) **backend file(s)**.

#### **Source files**

- Contain **local Tables**.
- They may have different formats (MS-Access, SQL server, Azure, Excel, text, html, ...).
- They are usually used **simultaneously** by **many** users, that access their Tables through their **frontend file(s)** and through the network.
- Each user usually accesses simultaneously to one (or more) **source file(s)**.

Notice that there is **no specific difference** between an MS-Access **frontend file** and an MS-Access **backend file**. Both are MS-Access files with a similar internal format, capacities and characteristics. The difference is therefore in what content do you store in them and how do you use them.

Notice also that a **backend file** is a particular case of a **source file**, when the source file **format** is MS-Access.


### **K.3.4 What are local Tables, linked Tables and source Tables?**

I now present the main characteristics of a **local Tables**, **linked Tables** and **source**







Tables:

### Local Tables

- Are **stored** in a **source file** that can have **database** format (MS-Access, SQL Server, Azure, ...) or **non-database** format (Excel, text, html, ...).
- In an MS-Access file, each **local** Table is shown in the “**Navigation Pane**” with the icon “”.
- The content of a given **local** Table can be managed (enter/modify/delete records) by opening the **source file** where it is **stored** and also by opening a **frontend file** with Tables **linked** to it.
- The properties of a given **local** Table (fields, field types, validation rules, ...) can be managed opening the **source file** where it is **stored**.
- Can have Relationships **with** referential integrity within **the same database source file**. These Relationships can be established/modified/removed opening the **database source file** where the **local** Table is **stored**.

### Linked Tables

- **Seem to be** a Table in an MS-Access **frontend file**, but in fact they are a **link** to its **source** Table, which is **local** Table in a **source file**. When I say that a **linked** Table is **linked** to its **source** Table it means that the **path** to, and the **format** of, the **source file** where the **source** Table is **stored** are recorded in the **linked** Table.
- Are shown in the “**Navigation Pane**” with a **specific icon** depending on the **format** of their **source file**. Some examples of icons are “” for MS-Access, “” for Excel, “” for html or “” for text **source file formats**.
- Opening the **linked** Table allows to **change** (enter, modify and/or delete) the records of its **source** Table.
- The **linked** Table properties (fields, field types, validation rules, ...) **cannot** be managed opening the file where it is placed: you have to do it in the **source file** where its **source** Table is **stored**.
- **Linked** Tables **cannot** have Relationships **with** referential integrity: only its corresponding **source** Table **stored** in the **source file** can have them. **Linked** Tables **can** have Relationships **without** referential integrity.

### Source Tables

- Are **Tables** that are linked from one (or more) **linked** Tables that are place in **frontend files**.
- The content of a given **source** Table can be **changed** (enter, modify and/or delete records) opening **any linked** Table (from its corresponding **frontend** file) that is linked to the given **source** Table.
- The **file** where a **source** Table is **stored** is called the **source file**.
- **Source files** can have **database** formats (MS-Access, SQL Server, Azure, ...) or **non-database** formats (Excel, text, html, ...).
- A **source** Table is always a **local** Table in the **source file** where it is stored.

When I say that a **source** Table is **stored** in a **source file** it means that **all the source Table data** and **all the source Table properties** are **stored** in the **source file**.

Notice that the links to the **source** Tables are absolute file paths, starting with the drive letter (e.g., “**D:**”, “**C:**”, etc.), followed by all the folder names, and finally the file name with the corresponding file extension. This means that if you change the name/and or

move a **backend** file, you will have to **refresh** its **link** (i.e., the file path and name) in **all** the **frontend** files that were using **source** Tables from that **backend** file. Notice also that if you move **both** a frontend field and a backend file together to another folder, you **also** have to **refresh** the **link** to the backend file, because the **link** is an absolute **file path** and **name**, and it has changed.

### K.3.5 How do I get a split database?


#### What is splitting a database?

**Splitting** the database means **separating** the Queries, Forms, Reports, VBA modules and Tables in **different frontend, backend files** (and/or possibly other **source files**, click *K.3.3*). MS-Access **frontend files** may contain Queries, Forms, Reports and VBA modules (and possibly some local Tables), while MS-Access **backend files** typically contain only **source** Tables (in MS-Access format) and **source files** contain **source** Tables with formats other than MS-Access. The **frontend files** with the Queries will usually have **linked** Tables (click *K.3.4*) linked to **source Tables stored in backend files** and/or other **source files**. These links to Tables in other files are called a “**linked Table**”.

#### **How do I automatically split one database file?**

If you have a “normal” MS-Access file with Queries, Forms, Reports, VBA code and Tables, there is an MS-Access command to **automatically split** it in one **frontend file** and one **backend file**. The **frontend file** will have **all** the Queries, Forms, Reports and VBA modules, and the **backend file** will have **all** the (**source**) Tables. The **frontend file** will have **linked Tables** to **all** the **source** Tables in the **backend file**.

If you **automatically split** an MS-Access database file, the file that you **split** will become the **frontend file** and a **new backend file** will be created. If you want to **keep** the original integrated file, you should **keep a copy** before doing the **split** operation.

To split an MS-Access file open it, click on “**Database Tools**” and then on the **Access Database**  icon. You will get a warning message explaining what the tool does, showing two buttons “**Split Database**” and “**Cancel**”: click on “**Split Database**”. You now get a file manager window where you can decide the name and location of the **backend file**. The default file name for the **backend file** is the current MS-Access file name adding to it the suffix “**\_be**”, and having the same “**.accdb**” extension. Once you have selected the **backend file** name and location, you click on the “**Split**” button. After a couple seconds you will see an informative window with an “**OK**” button, that you should click to remove it. MS-Access has now **left all** Queries, Forms, Reports and VBA modules in the **frontend file**, has **moved all** the Tables to the **newly created backend file**, and has **created linked** Tables in the **frontend file** to **all** the **source** Tables in the **backend file**. If you now **run** any of the Queries, Forms or Reports in the **frontend file** it will work as previously. If you now open in “**Datasheet View**” a (**linked**) Table in the **frontend file**, it will open normally, and you will be able to **view, enter, modify** and/or **delete** the records of the (**source**) Table, which is actually placed in the **backend file**.

#### **How do I manually split a database?**

You can manually create as many MS-Access **frontend** and **backend files** as you need. You can also manually create as many **source files** with other formats (e.g., Excel, text, ...) as you need. You can manually **copy** and/or **move** Tables, Queries, Forms, Reports


and VBA modules **between** MS-Access **files**. Doing this you can distribute your Tables, Queries, Forms, Reports and VBA modules across different files as you want. The only thing you are missing is how to manually **create linked Tables**. I explain this in the next section “*K.3.6 How do I create a linked Table?*”.

Notice that if you automatically split one MS-Access file as indicated in the previous question, all the Table links will be **automatically** created, and you do not need to do it manually.

### **K.3.6 How do I create a linked Table?**

**Creating a linked Table** consists of creating a **link** to a given **source Table** stored in a given **source file**.

A **linked Table** is somehow like a Form, in the sense that it allows to **enter**, **modify** and/or **delete** records from the **source Table** it is linked to, but it is **not** the Table itself. In a **linked Table** you **cannot** change the field names, delete/add fields nor change the properties of its **source Table**. If you open a **linked Table** in “**Design View**” you will **see** the Table’s properties, and you can even change them, but **you cannot save the changes**. If you **delete** a **linked Table**, you **do not** affect its **source Table** in any way.

You may **create linked Tables** by clicking on the “**External Data**” Ribbon name, and then on the **New Data Source**  icon. You will get a **first** pop-up menu, where placing the mouse over each of the menu-items shown will display an item-specific **second** pop-up menu containing the **actual options**. Clicking on one of the options will select the **type of source file** (MS-Access, Azure, Excel, text, ...) that you want. A dialog box will be shown, where you can select the following:

- The file **path** and **name** of the **source file**  
Clicking “**Browse...**” allows you to **select** the **source file** using a file selector box. You can alternatively type-in the file **path** and **name** to the **source file**, but this is usually cumbersome.
- The **type of operation**  
The available **options** for the type of operation depend on the **type of source file**, but they will always include “**Link to the data source...**” to create a **linked Table**: tick this option. Examples of **other** options are the two additional ones for an Excel file: tick “**Import the source data...**” to create a **local Table** (importing the data from the **source Table**) or tick “**Append a copy of...**” to **add** the data from the **source Table** to an **existing Table**.

When you are done, you click on “**OK**”. Depending on the **type of source file**, some dialog boxes will open to assist you in selecting the **source Table** and/or the **proper** format. For the case of a MS-Access file format, a new dialog box is shown, listing **all** the **names** of the Tables **stored** in the **source file**. You now **select** the **source Table names** that you want to **link** by **clicking** on each of them. Clicking on each **source Table** name will **toggle** it selected/unselected. **Selected** Tables are **highlighted** with **blue** background. The dialog box has the buttons “**Select All**” and “**Unselect All**”, that you can also use to faster select your desired set of **source Tables**. Once you have selected all the **source Tables** you want to **link**, you click on “**OK**”, and you are done.

You can also **create linked Tables** using the “**Add**” command of the “**Linked Table**

Manager” (click *K.3.10*).

### K.3.7 How do I refresh a Table link?

**Refreshing** a **Table link** consists of **linking** the **linked Table** to the corresponding **Table** in the corresponding **source file**. This is done, for example, after having corrected a **wrong source file name** or having **moved** the source file.

You **refresh** a **Table link** in either of the following ways:

- **Right-click** on the **linked Table name** in the “**Navigation Pane**” and then click on “**Refresh Link**” from the pop-up menu.
- **Open** the “**Linked Table Manager**” (click *K.3.10*) and **tick** the checkboxes of **all** the **Table link(s)** that you want to refresh. You then click on the “**Refresh**” button. The result of the refresh (“**Succeeded**” or “**Failed**”) for **each** **Table link** will be shown in the column “**Refresh Status**”.

Regardless of how you do it, if the refresh **fails**, you get one dialog box indicating that the refresh failed. You remove this dialog box by clicking on “**OK**” or its close icon “**X**”, and then MS-Access will open the “**Linked Table Manager**” (click *K.3.10*).

An **annoying** side effect of **refreshing** a link is that the corresponding **linked Table** is **unhidden** in the “**Navigation Pane**”. I think this may be an MS-Access **bug**. If you want it/them to remain hidden, you will have to **manually** hide them again.

### K.3.8 How do I convert a linked Table to a local Table?

Converting a **linked Table** to a **local Table** consists of **copying** all its **data** and **properties** (field types, validation rules, default values, ...) to the **local** file, removing the existing link, and removing all the Relationships with referential integrity that the **source** Table has in its **source** file.

To convert a **linked Table** to a **local Table**, you right-click over the linked **Table name** in the “**Navigation Pane**”, and then click on “**Convert to Local Table**” from the pop-up menu. Depending on the Table’s Relationships, you have three cases:

- The Table **does not** have any Relationship:  
Then, the **linked Table** is converted into a **local Table**, and the former **Table link** is deleted.
- The Table **has** at least one Relationship **with** referential integrity:  
Then, MS-Access shows a warning message indicating that the Relationship will be suppressed, that seems to imply that if you click on “**OK**” the **linked Table** will be converted to a **local Table**. However, the **linked Table** will **not** be converted to a **local Table** regardless of you clicking on “**OK**” or “**Cancel**”. This may be an MS-Access **bug**.
- The Table **has no** Relationship **with** referential integrity, and has **at least one** Relationship **without** referential integrity:  
Then, the command does **nothing**, and the **linked Table** is **not** converted to a **local Table**. This may be an MS-Access **bug**.

Therefore, if you want to **automatically** convert a **linked Table** to a **local Table** you

should **first** remove **all** its Relationships in the **source file**. You may afterwards create Relationships with referential integrity, but only with other **local** Tables.

You can **also** convert a **linked Table** into a **local Table** by **first deleting** the **linked Table** and then **copying** (if it is an MS-Access source file) or **importing** the **Table** from the **source file** (and if needed, renaming the Table). Remind that when you **delete** a **linked Table** you are **only** deleting the **link** in the **frontend file**, and the **source Table** in the **source file** remains totally unaffected.

### **K.3.9 How do I view a Table link?**

**Viewing** a **Table link** consists of **viewing** the file **path** and **name** of the **source file**, plus the **source Table name** in the **source file**.

If you place the mouse over a **linked Table name** in the “**Navigation Pane**”, MS-Access will show (in a small pop-up box) the file **path** and **name** of its **source file**, but it will **not** show the **source Table name** in the **source file**.



To **view** a **Table link**, you open the “**Linked Table Manager**” (click *K.3.10*). You then either **manually** locate the **linked Table** whose link you want to view, or you **automatically** search for it.

To **manually** locate a linked Table, you **unhide** the **linked Tables** of the **source file(s)** that you want to check. To **unhide** the **linked Tables** of a given **source file**, you just click on the “+” character on the leftmost side of the **source file row**.

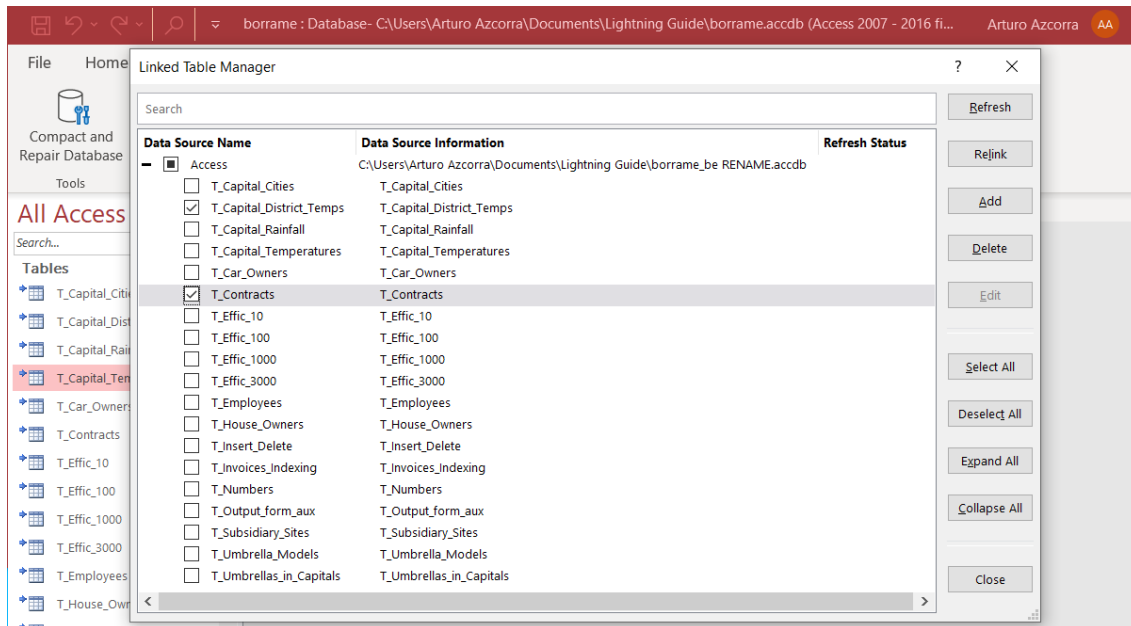
To **automatically** search for a **linked Table**, you type-in (in the search box at the top of the “**Linked Table Manager**”) a **string** and all the **linked Tables** whose name **contains** that **string** will be **unhidden**, while all the other ones will be **hidden**.

### **K.3.10 How do I view and manage Table links?**

You **view** and **manage** Table links with the “**Linked Table Manager**”. You open the “**Linked Table Manager**” in either of the following ways:

- **Right-click** on **any local** or **linked Table name** in the “**Navigation Pane**” and then click on “ **Linked Table Manager**” from the pop-up menu.
- **Click** on the “**External Data**” Ribbon name, and then on the **Linked Table Manager** “” icon.

Either way, the “**Linked Table Manager**” dialog box will be opened, as shown in the following screenshot:



The “**Linked Table Manager**” shows **each source file** that **stores** one (or more) **linked Tables** in a row called “**data source**”. The **same source file** may be shown in **several data source rows**, if you configured it in this way. Each **data source row** contains, left to right, a “+” or “-” character, a **checkbox**, the **name** of the **data source row**, and the **file path and name** of the **source file** in this **data source row**.

For **each data source row**, its “+/-” character and its checkbox work as follows:

- If you **click** on the “+” character, **all the linked Tables** that are **linked** in **this data source row** are **unhidden** and shown **below** the data source row, and the character becomes a “-”. **Each linked Table** is shown in **its own row**, all below the data source row. Each **linked Table row** contains, left to right, a checkbox, the **name** of the **linked Table**, the **name** of the **source Table**, and the “**Refresh Status**” (shown only after a “**Refresh**” operation).

If you **click** on the “-” character, **all the Tables** that are **linked** in **this data source row** are **hidden**, and the character becomes a “+”.

- If you **tick** the checkbox, the checkbox of **all the Tables linked** in **this data source row** become **ticked**.

If you **untick** the checkbox, the checkbox of **all the Tables linked** in **this data source row** become **unticked**.

**Clicking** on one of the **buttons** placed on the **right** side will do the following:

- **Refresh**  
**Refreshes** the links of all the linked Tables that are **ticked**. The result of the refresh (“**Succeeded**” or “**Failed**”) for **each Table link** will be shown in the column “**Refresh Status**”.
- **Relink**  
**Changes** the **source Table** and/or the **source file** of the **linked Tables**.

The result of clicking “**Relink**” depends on what **linked Tables** are **ticked**, as follows:

1. If **some**, but **not all**, the **linked Tables** from a **data source row** are **ticked**, a dialog box is opened where you can type-in the **name** of a **different source Table** (from the **same source file**). MS-Access will sequentially open **one** dialog box for **each linked Table** that was **ticked**. The name of the **linked Table** being relinked is shown in the heading of each dialog box.
  2. If **all** the linked Tables from a **data source row** are **ticked**, a file selector box is opened: you select the **source file** that you want and then click on “**OK**”. The result now depends on the **selected source file** being the **same as**, or **different from**, the **current source file**:
    - a. If both files are the **same**, the result is the one in point **1** above: MS-Access will sequentially open **one** dialog box for **each linked Table** that was **ticked** (in this case, for all of them).
    - b. If both files are **different**, a dialog box is opened asking if you would like to **relink** the selected Tables. Be **careful** because the **default** option is “**Yes**”, and you **most likely** want to click on “**No**”. If you click on “**No**”, MS-Access will **automatically** relink **each linked Table** to the **source Table** having the **same name**. However, if you click on “**Yes**”, MS-Access will behave as in points **1**) and **2a)** above, painfully **asking one by one** for the **name** of **each and every source Table** that you want.
- **Add**  
**Adds** a **new data source row**, with a number of **linked Tables**. Clicking on “**Add**” opens a dialog box where you type-in the **name** for this **data source row**, select what is the format of the **source file** among four options, and click on “**Next**”.  
 You then click on “**Browse...**” to **select** the **source file** using a file selector box. You can alternatively type-in the file **path** and **name** to the **source file**, but this is usually cumbersome. You may also type-in the password for the source file, in case it is password protected. When you are done you click on “**Finish**”.  
 A new dialog box is shown, listing **all** the **names** of the Tables **stored** in the **source file**. You now **select** the Table **names** that you want to **link** by **clicking** on each of them. Clicking on each Table name will **toggle** it selected/unselected. **Selected** Tables are **highlighted** with **blue** background. The dialog box has the buttons “**Select All**” and “**Unselect All**”, that you can also use to **faster select** your desired **source Tables**. Once you have selected all the **source Tables** that you want to link, you click on “**OK**”, and you are done.  
**Each linked Table** is **created** with the **same name** as its **source Table**. If the **source Table name** already exists in the **frontend file**, then a number (“**1**”, “**2**”, ...) is appended to the **linked Table name**. You may change the **name** of any **linked Table** as you do with a **local Table** (click *B.4.1.8*).
  - **Delete**  
**Deletes** all the **linked Tables** that are **ticked**. Remind that this **does not** delete the **source Table**.



- **Edit**  
Edits an existing **data source row**. Clicking on “**E**dit” opens a dialog box for the **data source row** that was **ticked**. There you can change the **name** of the **data source row**, the file **path** and **name** of its **source file** and the **password** of its source file. When you are done, click on “**S**ave”.
- **Select All**  
Ticks the checkboxes of **all data source rows** and **all linked Tables**.
- **Unselect All**  
Unticks the checkboxes of **all data source rows** and **all linked Tables**.
- **Expand All**  
Unhides **all** the linked Tables from **all data source rows**.
- **Collapse All**  
Hides **all** the linked Tables from **all data source rows**.
- **Close**  
Closes the “**Linked Table Manager**” dialog box. Clicking on the close icon “**X**” also closes the “**Linked Table Manager**” dialog box.

If you have many **linked Tables**, you can search for a specific one using the search box at the top of the “**Linked Table Manager**”. If you type-in a **string** in the search box, all the linked Tables whose name **contains** that string will be **unhidden**, while all the other ones will be **hidden**.

Some situations (e.g., cancelling a link operation) may cause the “**Linked Table Manager**” to be in an inconsistent state and show strange information. If this happens, close it and open it again and most likely it will come back working well.

### **K.3.11 How do I delete a linked Table?**

If you just want to delete (in the **frontend file**) the **link** to the Table, and not its corresponding **Table** in the **source file**, you can do it in the following ways:

- **Right-click** on the **linked Table name** in the “**Navigation Pane**” and then click on “**D**elete” from the pop-up menu.
- **Select** the **linked Table** in the “**Navigation Pane**” and then press the “**Supr**” key.
- **Open** the “**Linked Table Manager**” (click *K.3.10*) and **tick** the checkbox(es) of **all** the **linked Table(s)** that you want to delete. You then click on the “**D**elete” button.

Either way you get a dialog box with indicating that this will **only** delete the Table **link**, and it **will not** delete the actual **Table** in the **source file**. If you click on the “**Yes**” button, the Table link is deleted. If you rather click on “**No**”, the delete operation is cancelled.

If you actually want to delete the **source Table** in the **source file**, you have to open the **source file** and delete the Table there as you would normally delete any Table in the corresponding application (MS-Access, Excel, ...) that manages that specific **source file**.

### **K.3.12 How do I manage Relationships with linked Tables?**

**Each database file** (regardless of being frontend or backend) has **its local** and **specific**



currently configured **Relationships**. From any **given** database file you **cannot view**, nor do any operation, on the Relationships that are configured on **another** database file.

You **can configure** Relationships **without** referential integrity between **local Tables**, between **linked Tables** and between one **linked Table** and one **local Table**. However, the usefulness of Relationships **without** referential integrity is quite low and my advice is you do not configure them.

You **can only configure** Relationships **with** referential integrity between **local Tables**.

If you want to establish a Relationship, you may click “*D.9 How do I create and configure my Table Relationships?*”.

### **K.3.13 Can I change the file path or name of a source file?**

Yes, you can do it, but you then have to **relink all** the corresponding **linked Table links** to this **source file** in **all the corresponding frontend files**.

To **update all** the Table **links** when you have changed the file **path** or **name** of a **source file** you can:

- Update the Table **links** in the different **types** of **frontend files** that you have, and then send the updated **frontend files** to the corresponding users.
- If the users are familiar enough with MS-Access, you can send them a notification with the **new file path** and **name** of the **source file**, so they do the update themselves.

A good design option is creating the Table **links** to **shortcuts** or **hard links** (in the file system) to the actual **source files**, as follows:

- You can create **all** your Table **links** to a windows **shortcut** to each corresponding **source file**. If you do this, when you change the file **path** or **name** of a **source file** you only need to update it in the corresponding **shortcut**, and then everything will work the same.
- You can create **all** your Table **links** to a windows **hard link** to each corresponding **source file**. If you do this, you can change the file **path** or **name** of all the hard links, except the one used in the frontend files, and everything will work with no changes.

### **K.3.14 What are the options to lock records in a shared database?**

**Coherence** of the stored data is achieved by “**locking**” the database records to avoid that they are **modified in parallel** by two, or more, users. MS-Access supports three locking strategies that you can set in the option “**File**→**Options**→**Client Settings**→**Advanced**→**Default record locking**”, as follows:

- “**No Locks**”
- “**All Records**”
- “**Edited Record**”

Since concurrent users may modify records simultaneously, we need some way to guarantee that the stored data will be coherent, and a record will not contain some of its field values coming from one user, and other values coming from a different user, both of which were editing the record at the same time. The way to prevent this is by “locking” a record that is being edited by a user, so no other user can edit it simultaneously. There

are three ways in that the system can implement this “locking”, which correspond to the three possible values of the option “**Default record locking**”, as I will now explain:

#### “No Locks”

MS-Access will **allow several users** to be **editing the same Table record**. However, when they attempt to **save** the record, **only the first user** will be allowed. The **other users** that were editing the record will get an **error message** when they try to save it. Each of the other users can decide to discard her/his edits or copy his/her values to the clipboard or to an external file. Then each of the other users may attempt to edit the record again, this time over the value that the **first** concurrent user successfully saved. This locking strategy is called “**optimistic**” locking. This is the default value in MS-Access.

All users can **view** records from any Table.

#### “All Records”

MS-Access will **only allow one user** to be **editing several records** from **the same Table**. The moment **one** user opens **one** record for **editing**, the **whole Table** is **locked**, and no other user is allowed to edit any record in this Table until the first user has finished editing the record.

All users can **view** records from any Table.

#### “Edited Record”

MS-Access will **only allow one user** to be **editing each given Table record**. The moment **one** user opens **a given** record for editing, **that specific record** is **locked**, and no other user is allowed to edit that record. Other users may edit other record in this Table, as long as the record is not already being edited by some other user. This locking strategy is called “**pessimistic**” locking.

All users can **view** records from any Table.

#### Conclusion

My advice is you configure “**Edited Record**” locking (i.e., **pessimistic** locking).

### **K.3.15** What is the delay of network access to a database?

When users access a database through a network there are two types of computers. The computer of each user is called the “**client**” and the computers where **source files** (click K.3.3) are located are called the “**servers**”. Each **client** computer typically contains the user’s **frontend files** and the **server** computers contain the **backend** files.

When a user **runs** a Query from a **frontend file**, the **client requests** the required information to the **servers**. The required information are the corresponding **fields** and **records** from **all** the **innermost** SQL operations of the Query. These **innermost** operations are the ones that have a **Table** as their **input** record-list. The information that is requested from the client to the server is **not** the whole Table, for efficiency reasons, and only the **required records** and **fields** are requested. The **requested information** is typically **only** the records that produce **True** in the “**WHERE**” **Boolean** expression from each of the innermost **Select** operations, and from those records, **only** the fields that are used in the “**SELECT**” **expressions**.

When the client **requests** information from a **server**, getting it takes some time usually

called “**delay**”, and this **worsens** the response time of running the Query. This **delay** has two very different components:

- **Transit delay**  
This is the time it takes the “**request**” **packet**<sup>104</sup> to go through the network from the **client** to the server. This is also (approximately) the time it takes a **packet** containing (part of) the requested information to go through the network from the **server** to the **client**. The **transit delay** depends mainly on how far away the client and the server are.
- **Transmission delay**  
This is the time it takes to **send** **all** the packets containing **all** the requested information. The **transmission delay** is computed as the amount of information that has been requested divided by the network throughput in the path from the server to the client.

The **total** delay to receive the data is **twice** the **transit** delay (which is called the “**round trip**” delay) plus the **transmission** delay.

If you want to visualize this, imagine that you are a skiing instructor (the request packet) at mountain top, and you notice that all your trainees (the requested data) took the wrong slope. You need to go fetch your trainees and bring them back to the top, to take the correct slope. It takes you some time to go down the slope, to meet the trainees at the end of the slope: this is **one transit delay**. You tell all your trainees to mount on the chairlift to go back to the mountain top. The trainees taking the **first** chairlift (the data packet) take some time to arrive to the top: this is the **second transit delay**. The trainees that are **first** to arrive still have to **wait** for **all the other** trainees to reach the top, chairlift after chairlift (packet after packet). This **waiting** time (the transmission delay) is the number of trainees divided by the throughput of the chairlift: if you have 50 trainees and the chairlift takes 200 people per hour<sup>105</sup> it will take 15 minutes for all of them to arrive at the top. As you may see, the total waiting time is **two transit delays** plus the **transmission delay**.

### K.3.16 Why should I make temporal Tables local?

If you use **temporal** Tables, and a **distributed** database (click [K.3.1](#)), always make your temporal Tables **local** (click [K.3.4](#)) in the **frontend files** (click [K.3.3](#)). This will not only prevent possible collisions between concurrent users, but will **also** improve the performance of the database, because local operations are always faster than operations across the network (click [K.3.15](#)).

## K.4 What Query design principles should I follow?

You may click:

- “[K.4.1 How do I write my SQL Queries?](#)”
- “[K.4.2 Why should I incrementally run my SQL code while I write a Query?](#)”

---

<sup>104</sup> A “packet” is a block of information sent through a network. A packet usually has a maximum size of 1,500 bytes.

<sup>105</sup> These figures are **not realistic** but serve better to illustrate this question.

- “K.4.3 How do I write readable (maintainable) SQL Queries?”
- “K.4.4 Why should I disable changing Table data from Query results?”
- “K.4.5 Why should I qualify all the outermost output field names of my Queries?”
- “K.4.6 When should I remove duplicate records?”
- “K.4.7 Why should I enclose the outermost Union operation in a Select operation?”
- “K.4.8 Why should I restrict the usage of “INNER JOIN”?”
- “K.4.9 Why should I avoid using “SELECT \*”?”
- “K.4.10 Why should I avoid using the same name of an input field name for an output expression?”
- “K.4.11 When are “SELECT” expressions evaluated along Query processing?”

### K.4.1 How do I write my SQL Queries?

SQL Queries are written in plain text. My advice is that you write your SQL code using the plug-in “**Access SQL Editor**” (click F.5).

When writing a **consulting** Query, the first SQL operation (i.e., the outermost SQL operation) in the Query code **must** be either a **Select operation** (click F.7), a **Union operation** (click F.9) or a **Transform operation** (click F.10). It **cannot** be a **Join operation** (click F.8).

Although it is possible to have a **Union** operation as the first expression in the Query, I strongly advise you **do not do it**, because if you do, you **cannot customize the field formatting**. The solution is just enclosing the **outermost Union** operation in a **Select** operation (click K.4.7).

Every SQL Query **should** end with a **final** semi-colon “;”. However, MS-Access **does not** enforce this, and therefore, if you forget to add the **final** semi-colon “;” your Query will work fine.

Let me show you an example of a simple **Select** Query.

```
SELECT Capital, District
FROM T_Capital_Temps ;
```

And the following is an example of a simple **Transform** Query.

```
TRANSFORM Avg(Temp_max)
SELECT Capital
FROM T_Capital_Temps
GROUP BY Capital
PIVOT Quart ;
```

### K.4.2 Why should I incrementally run my SQL code while I write a Query?

SQL operations **appear** to be simple, but **they are not**. It is extremely easy to think your code is doing something, while it actually doing something **slightly different**. The presence of duplicate records, **Nulls** and other situations frequently causes your Query code to do something **slightly different** to what you intended.

It is therefore very good practice to write your Query code “**inside-out**”, and **progressively run** your Query code to debug it **at the same time** as you write it. Doing this will save you considerable time because you will detect errors earlier in the process. You will also detect earlier **wrong assumptions** that you had made over your data and will then help you redesign your Query to produce your intended results.

By “inside-out” I mean that you start writing the **innermost** Select operations extracting data from your Tables, and you **progressively** combine them with **Join** and **Union** operators, until you build the complete Query.

While you write your Query code, in order to run the specific SQL operation that you want at any given moment you will have to **comment/uncomment** (click *J.3*) your code.

As you have noticed, the technique used to **run** and **debug** your Query **while you write it** is basically the same used to debug the complete Query. If you want to know more about this, you may click “*Part J. Debugging my SQL Queries*”.

### **K.4.3 How do I write readable (maintainable) SQL Queries?**

I provide here a few suggestions to improve the readability, and therefore the maintainability, of your SQL Queries:

- **Align all the clauses of each SQL operation**
- **Align the Join operator and the “ON” clause with its enclosing Select**
- **Indent the two input record-lists of Join and Union operators**
- **Indent the interior SQL operations in your Query code**
- **Align all the input record-lists in associative operations**
- **Place the commas “,” at the beginning of each line**
- **Place parentheses of SQL operations to allow their comment/uncomment**
- **Add comments throughout your SQL code**
- **Clearly explain the Query output**
- **Identify auxiliary Queries**
- **Trust your current code**
- **Prefix each and every Table name with “T\_”**
- **Use auxiliary Queries when needed**
- **Use Union operations as much as possible**

I now explain each of these points in more detail.

#### **Align all the clauses of each SQL operation**

All the clauses (“**SELECT**”, “**FROM**”, “**WHERE**”, “**ORDER BY**”,...) of **each** given SQL operation should be **aligned**. This improves readability a lot, because it visually shows

what clauses belong to what SQL operation. For example<sup>106</sup>:

```

TRANSFORM    5*Out_fld
SELECT      Log(2*Out_fld) AS Nonsense_1
FROM        T_Capital_temps
GROUP BY    Len(Capital), 2*Cal_Year
ORDER BY    2*Cal_Year DESC, Len(Capital)
PIVOT      "Q" & Quart ;

```

### Align the Join operator and the “ON” clause with its enclosing Select

When writing a **Join** operation, it must be enclosed in a **Select** operation, and the best practice is to align the Join operator(s) and the “**ON**” clause with the other clauses of the enclosing **Select** operation. For example<sup>107</sup>:

```

SELECT T_House_Owners.Address, Cars.Car, Non_capital_cities, Capital
FROM
    T_House_Owners
    ,
    (
        SELECT Car
        FROM T_Car_Owners
        WHERE ID > 0
    ) AS Cars
    ,
    F_Select_nested AS Q
    ,
    (
        SELECT Capital
        FROM T_Capital_Cities
    ) AS Capitals

```

Notice how the **Cross-Join** operator “,” are aligned with the clauses “**SELECT**” and “**FROM**” of their enclosing **Select** operation.

Another example<sup>108</sup>, now with an “**INNER JOIN**”:

```

SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
INNER JOIN
    T_Car_owners AS Cars
ON Houses.ID = Cars.ID ;

```

Notice how the “**INNER JOIN**” operator and its “**ON**” clause are aligned with the clauses “**SELECT**” and “**FROM**” of their enclosing **Select** operation.

### Indent the two input record-lists of Join and Union operators

Indenting (i.e., adding spaces at the beginning of each line) the two input record-lists of **Join** and **Union** operators, **visually distinguish them** from the **Join** or **Union** operator itself. You may see the two examples of **Join** operators I have just shown above, where their input record-lists are indented. Let me show an example<sup>109</sup> with the **Union** operator:

<sup>106</sup> This is a subset of the Query “**F\_Transform\_syntax**” from file “**Company\_Database.accdb**”.

<sup>107</sup> This is the Query “**F\_Join\_nested\_Cross**” from file “**Company\_Database.accdb**”.

<sup>108</sup> This is the Query “**F\_Join\_Inner\_1**” from file “**Company\_Database.accdb**”.

<sup>109</sup> This is a subset of the Query “**F\_Union\_syntax**” from file “**Company\_Database.accdb**”.

```

SELECT Capital, Cal_Year, Rainfall
FROM T_Capital_Rainfall
UNION ALL
SELECT ID, Name, Len(Car)
FROM T_Car_Owners

```

### Align all the input record-lists in associative operations

When you have a series of Cross-Join, “**UNION**” or “**UNION ALL**” operations, recall that they are associative. For this reason, it is a good practice to show **all their input record-lists aligned**. Let me show an example of the **Union** operator:

```

SELECT Capital, Cal_Year, Rainfall
FROM T_Capital_Rainfall
UNION ALL
SELECT ID, Name, Len(Car)
FROM T_Car_Owners
UNION ALL
SELECT Capital, District, Temp_Min
FROM T_Capital_Temps
UNION ALL
SELECT Capital, District, Temp_Min
FROM T_Capital_Temps

```

### Indent the interior SQL operations in your Query code

Indenting (i.e., adding spaces at the begging of each line) the interior SQL operations **visually shows the structure** of the Query in terms of its different internal components. This makes your SQL code much more readable and will be much easier to debug and maintain. You **should** progressively indent the interior SQL operations to facilitate understanding the structure of the Query code. Proper code indenting is a **must** in any programming language. One example<sup>110</sup> of SQL code indenting is:

```

SELECT City AS Non_capital_cities
FROM
(
  SELECT City
  FROM
    (SELECT City
     FROM T_Subsiadiary_sites
     WHERE City <> "Washington" )
  WHERE City <> "Paris"
)
WHERE City <> "Madrid"

```

Notice how **each** interior **Select** operation is indented in respect to its enclosing **Select**. Notice also that I have colored matching parentheses for your convenience.

### Place the commas “,” at the beginning of each line

When you split in **several lines** a list of elements separated with **commas** (e.g., “**SELECT**” **expressions**, “**GROUP BY**” **expressions**, “**IN**” **list**, ...), place the separating comma “,” of the elements between lines at the **beginning** of each line, instead of at the **end** of each line. This may look strange at first, but I can tell you it will save you considerable time preventing to forget the comma (or having two commas). Look at the

<sup>110</sup> This is the Query “**F\_Select\_nested**” from file “**Company\_Database.accdb**”.

following example<sup>111</sup>:

```

SELECT TOP 80 PERCENT
    Len("City_" & Capital)                AS Out_fld
, 2 * Len(Out_fld)                        AS Nonsense_2
, 3 * Sum(3 + Temp_Max)                   AS Nonsense_3
, 4 * (1+(9+Temp_min) + Max(5*Temp_Max)) AS Nonsense_4
, 5 * Avg(Len(Out_fld))                   AS Nonsense_5

```

Notice how **each** comma “,” that is separating two elements in consecutive lines is placed at the **beginning** of the line, instead of at the **end** of the line.

### Place parentheses of SQL operations to allow their comment/uncomment

You debug your SQL by commenting/uncommenting whole SQL operations (click *J.3*). It is therefore very convenient to place the parentheses that enclose **each** SQL operation in a way that allows for the automatic commenting/uncommenting (see the comment/uncomment commands in *F.5.5.1*) of the operation. This typically implies placing **each** enclosing parenthesis in its own line, except for the enclosing parentheses of a **single Select** operation without an “**AS**” clause, which can be in the same lines as the code of the **Select** operation. The following example<sup>112</sup> shows how **not** to place parentheses:

```

( (SELECT ID AS New_1, Name AS New_2, Address AS New_3
  FROM T_House_Owners )
UNION
  SELECT ID, Name, Len(Car)
  FROM T_Car_Owners
UNION
  SELECT Capital, District, Temp_Min
  FROM T_Capital_Temps )
UNION ALL
(SELECT Capital, Cal_Year, Rainfall
  FROM T_Capital_Rainfall )

```

If you now wanted to run **only** the **first Select**, or the **third Select** or the **first block of Unions**, you **cannot** add automatic comments to do it, and you have to **manually** edit your code!

<sup>111</sup> This is a subset of the Query “F\_Select\_w\_group\_by\_aggreg\_Syn” from file “Company\_Database.accdb”.

<sup>112</sup> This is the Query “F\_Parentheses\_1” from file “Company\_Database.accdb”.



However, if you rather write the parentheses in the following way<sup>113</sup>:

```
(
  (SELECT ID AS New_1, Name AS New_2, Address AS New_3
   FROM T_House_Owners )
 UNION
  SELECT ID, Name, Len(Car)
   FROM T_Car_Owners
 UNION
  SELECT Capital, District, Temp_Min
   FROM T_Capital_Temps
)
UNION ALL
(SELECT Capital, Cal_Year, Rainfall
 FROM T_Capital_Rainfall )
```

you can now do **automatic** comment/uncomment to isolate **any part of the Query** that you want to run individually for debugging purposes. An added advantage is that parentheses matching is also much clearer. The only disadvantage is that the Query becomes longer in terms of code lines, but on my view, the advantages outweigh this.

### Add comments throughout your SQL code

It is an **absolute must** to write **comments** within your Query code. Comments are text that you write in plain English, within the SQL code, to explain what the SQL code means. Comments are **essential** every time that you want to **review** a Query, to facilitate **understanding the logic** of the Query code. An SQL comment can be written within any SQL line of by placing it after **two consecutive hyphens**. This means that **any text after two consecutive hyphens** is ignored for the purpose of SQL processing. Let me show you an example of a Query with comments (highlighted in green so you see them better):

```
-- This Query produces a cross table of average maximum temperatures by
-- quarters
-- The expression after the "TRANSFORM" clause computes the average max
-- temperature for each quarter. "Avg()" is an SQL aggregate function.
TRANSFORM Avg(Temp_max)
-- This is the "SELECT" clause of the Transform operation
SELECT Capital -- This is the "SELECT" output field of the Select operation
FROM T_Capital_Temps -- This is the input record-list
GROUP BY Capital -- This is the "GROUP BY" expression ("Capital")
-- The PIVOT expression (in this case "Quart") computes the "PIVOT"
-- field names of the Transform operation
PIVOT Quart ;
```

When you write a Query everything is in your head and may **seem** obvious. However, six months later when you need to fix a bug, or you want to modify the Query, you look at the code and most likely you do not understand it, as if somebody else had wrote the code! It is therefore essential to add **clear comments** throughout the code, explaining why the code is like that. Do not mind if they look too verbose: they have to be as verbose as it is needed to **clearly explain** each SQL operation and each complex expression in your Query code.

### Clearly explain the Query output

At the begging of each Query, add an overall comment **clearly** explaining what the **output** record-list is and what the **precise** meaning of each **output** field is. An example

---

<sup>113</sup> This is the Query “F\_Parentheses\_2” from file “Company\_Database.accdb”.

of a bad, unprecise, comment may be:

```
-- Six-month large unpaid invoices and time unpaid.
```

An example of a better, **more precise**, comment would be:

```
-- Invoices with issue date in the six calendar months prior to the
-- current month that are unpaid and have an amount (without VAT)
-- larger or equal than 1,000 dollars.
-- The field "Time_unpaid" contains the number of calendar days between
-- the reception date of the invoice and the current date.
```

### Identify auxiliary Queries

In every Query that is used by other Queries and/or in drop-down menus, add at the very beginning a comment indicating this. In this way, you **avoid** happily modifying its functionality because of being unaware that modifying it will cause **side effects**! Before modifying its functionality, you must check (using a dependency checker tool, click *I.3*) what are the database objects that depend on this Query and analyze one by one if your intended modification will have a side effect or not. If you want to know more about side effects of Query modifications, you may click “*I.6 What are the side effects of modifying my Queries?*”.

### Trust your current code

When you fix a bug or modify the functionality of one of your Queries, it is very frequent that you find it unnecessarily complex and/or inefficient, and you feel there is a simpler/faster way to code the Query. You should **definitely refrain** from happily changing the code of a test and proven Query. First of all, because a test and proven Query is a **treasure**, and if you change it you risk losing this value. Second, because **most likely** the Query code **must be like it is**, and it is not likely that you found a way to make it better in just a glance. Before trying to make a Query code simpler, faster or better in any way, you must **carefully read all the comments**. This will allow you to **really** understand the Query code, and only after having done this, you should make a careful decision on whether it is really possible to improve the Query code.

### Prefix each and every Table name with “T\_”

This allows you to very easily distinguish Table names from Query names in your SQL code. This distinction is important in a number of situations. Adding the prefix “T\_” only makes your Table names two characters longer and does not reduce the meaningfulness or the rest of the name. The advantages of doing this are clearly worth the small effort required.

### Use auxiliary Queries when needed

Using an auxiliary Query (i.e., a Query that is invoked within a Query) is very useful in the following cases:

- When you want to make sure you are doing **exactly the same Select** operation over your Tables, for efficiency reasons. If you want more detail on this, you may click “*K.7.3 How do I design faster Select operations over Tables?*”.
- When your Query code becomes too long to debug properly (and/or exceeds the 65,535 character limit of the plug “**Access SQL Editor**”, in case you are using it). As a rule of thumb, a Query longer than 300 lines may be worth dividing in two or more

auxiliary Queries.

- When inside your Query you are using several times (e.g., three or more) the same fragment of SQL code that is reasonably large (e.g., twenty or more lines). Replacing that fragment by an auxiliary Query name that contains the SQL code in the fragment improves the readability of the Query. An added advantage is that it is likely that you will also use this auxiliary Query in other related Queries.

#### Use Union operations as much as possible

On many occasions, the **output** record-list of the SQL operation that you are writing is logically composed of several sets or records, each of which arise from a specific logic of the Query code. In all such cases, it is very useful to write the SQL operation as a **Union** operation, where each set of records is produced by each of the **Select** operations bound by the **Union** operations. Notice that **Union** operations are extremely good both for code readability and for ease of debugging.

#### K.4.4 Why should I disable changing Table data from Query results?

Because **changing** Table data from Query results in “**Datasheet View**” creates a **great risk** of the user **unwillingly** changing the Table data. If this happens, data in your Tables becomes wrong, and the results returned by Queries will also be wrong.

My advice is that you **disable** the feature of changing Table data from Query results in “**Datasheet View**”. This is done by setting the “**RecordsetType**” property (of the Query as a whole) to “**Snapshot**” in **all** your Queries. If you want to set Query’s properties, you may click “*B.7.2 How do I configure a Query’s “Property Sheet” in “Design View”?*”.

If for some reason you want/need to keep the “**RecordsetType**” property with its default “**Dynaset**” value (or even set it to the riskiest “**Dynaset (Inconsistent Updates)**” value), you can use the following **trick** to disable changing Table data from Query results in “**Datasheet View**”: **never** use **one field name** as an outermost “**SELECT**” **expression** of a Query and **always** use an **expression**. In case you actually want to have the value of one Table field in a Query result, you can **create** (in the outermost **Select**) a “**SELECT**” **expression** that does not modify its value: if it is a numeric-like field you can just add 0 (i.e., “**Field\_name+0**”); if it is a **Short Text** field, you can just append the **zero-length string** (i.e., “**Field\_name & ""**”).

This trick works because it is **only** possible to change Table data from Query results in “**Datasheet View**” in the following cases:

- An outermost “**SELECT**” **expression** of a Query corresponds **exactly** to **one** Table field (even if the Table field name has been changed): this allows to **change** the values of this Table field from the Query results in “**Datasheet View**”.
- **Each and every** outermost “**SELECT**” **expression** of a Query correspond **exactly** to **one** Table field (even if the Table field name has been changed): this allows (in addition to changing any Table field value) to **enter**, **modify** and/or **delete** Table **records** from the Query results in “**Datasheet View**”.

One of the relevant advantages of databases over spreadsheets is that databases **separate** the **data**, the **data processing** code, and the **results**. This separation prevents the user from mistakenly modifying the data and/or the data processing code while viewing data

or viewing results. In a spreadsheet the **data** (cells with values), the **data processing** code (formulas inside cells) and the **results** (cells with formulas) are all mixed up. This is very convenient, because you can arrange them in the layout that you want, and it is very easy to modify the data entry and/or the data processing. However, it is also **very risky**. In a database the data (**Tables**), the data processing code (SQL Query code) and the data viewing (Query results in “**Datasheet View**”) are perfectly separated.

Having the **data** completely separated from the **data processing** is one of the strong advantages of databases over spreadsheets. If database users can modify Table data from the Query result, this creates a **huge risk** of unintentional database data corruption.

This is why my strong advice is that you **always** disable Table editing from Query results, as I have indicated along this section.

#### **K.4.5 Why should I qualify all the outermost output field names of my Queries?**

Qualifying the **outermost output** field names of a Query prevents (**almost always**) that your Query **field formatting is lost** each time that you do a modification on the SQL code of the Query.

Doing the field formatting of a Query in its “**Datasheet View**” (click *H.6*) is essential to show properly formatted values, but it requires some work. Repeating this formatting work **over and over** each time that you do an adjustment to your Query code is something **you want to avoid!**

Qualifying the **outermost output** field names of your Query is something you only do once (when you write your Query) and the benefit of not losing the field formatting is **clearly** worth it. Notice that if you modify the SQL code of the Query **outermost output** fields, then you will lose the field formatting, but this is only **one** small case among the **many** other modifications you can do in your Query where you **do not lose your formatting**.

Remind that qualifying a field name consists of prefixing each field name with the name of the SQL operation where the field name is defined (click *C.2.2*). This implies that you have to assign a name to the **input** record-lists of your outermost **Select** operation or **Transform** operation. The **input** record-lists are either **one** (either another **Select** operation or a set of associative **Union** operation) or **two** (two **Select** operations in a **Join** operation), so the work of assigning them a name is really low.

#### **K.4.6 When should I remove duplicate records?**

Each time that you are designing an SQL operation you have two cases:

- If you want to **allow duplicate output** records **or** you are **completely sure** that the **output** record-list will not have duplicates because of the Table/Query design, then **do not add** clauses to remove duplicates to the SQL operation you are designing.
- If you are **reasonably** sure that you **do not want duplicate output** records **and** you are **not completely sure** that the Table/Query design prevents them, then you **should add** clauses to remove duplicates to the SQL operation you are designing.

Duplicates are acceptable in some cases and removing them would cause wrong Query results. You should **first** make reasonably sure that you **do not want duplicates**.

Duplicates are sometimes **prevented** by the Table and/or Query design. Therefore, in case that the design already prevents duplicates, you should **not** add clauses to **remove** them because this slows down your Queries unnecessarily.

Let me show an example where the Table/Query design **guarantees** that there are no duplicates. Imagine you are doing a **Select** over an **input** SQL operation, and its **output** fields are its **input** fields and its **output** records are just some of its **input** records. This is, the added value of this **Select** operation is to suppress some records from its **input** record-list. In case the **input** record-list is **duplicate-free**, it is **guaranteed** that the **output** record-list will also be duplicate free. For example, the **input** record-list would be duplicate-free in the following cases:

- It is a **Union** operation with the “**UNION**” operator.
- It is a **Select** operation with the “**DISTINCT**” clause.
- It is a **Select-total\_aggreg**.
- It is a **Select-group\_by\_aggreg** where **each and every** of the “**GROUP BY**” expressions are, **as such**, an **output** field. Notice there may be other output fields.
- It is a **Select** operation over a Table and its **output** fields include **as such** (i.e., not as part of an expression) the fields of the **Primary Key** and/or the fields of a **candidate Key** of the Table.
- It is a Query name whose outermost SQL operation falls in either of the cases above.

If you **actually want** to remove duplicate records, you can use the “**UNION**” operator or the “**DISTINCT**” clause<sup>114</sup>. Remind that this operator and clause will introduce some delay in your Query responses (notice MS-Access needs to check for duplicates even if there are none), so use them only **in case you need to**.

If you want more detail on this, you may click “*K.7.2.1 Why should I use “DISTINCT”, “UNION” and “ORDER BY” only if needed?*”.

#### **K.4.7 Why should I enclose the outermost Union operation in a Select operation?**

Because MS-Access **does not allow** to open a **Union** operation Query in “**Design View**”, and therefore, you **cannot** configure the **formatting** of this Query fields. A **Union** operation Query is a Query where the outermost SQL operation is a **Union** operation.

You will most likely want to format the fields of your Query, so the results are properly shown. The solution is very simple, and it consists of **enclosing** your outermost **Union** operation **in a Select operation**. Doing this will allow you to open the Query in “**Design View**”, and therefore, to configure the formatting of the Query result columns shown in “**Datasheet View**”.

Enclosing the outermost **Union** operation in a **Select** operation has the added advantage of allowing you to **sort** the **output** record-list with the “**ORDER BY**” clause.

---

<sup>114</sup> You can also use the “**DISTINCTROW**” clause, but my advice is you do not use it (click *F.7.8*).



#### K.4.8 Why should I restrict the usage of “INNER JOIN”?

Because “**INNER JOIN**” will not output records that do not match its “**ON**” *Boolean* expression. This **may seem ridiculous**, because this is exactly what “**INNER JOIN**” is expected to do, but it has **tricky** implications. On **many** occasions, you give for granted that the two **input** record-lists of the “**INNER JOIN**” will have the matching records that you expect. However, if one of the two **input** record-lists does not work as expected, and it does not contain the expected matching records, this will destroy the records from the other **input** record-list that are now not matching. This error may be very difficult to detect.

It is a **much safer practice**, whenever possible, to use an **Outer-Join** (e.g., a “**LEFT JOIN**”) instead of an “**INNER JOIN**”. In this way, if one of the **input** record-lists is missing, by mistake, some records, the **Outer-Join** will show **Null** in the corresponding fields of its **output** record-list. **Nulls** cause some effect on the Query much more likely than missing records. Therefore, it is **much preferable** that a possible mistake **creates Nulls** rather than **suppresses records**.

My advice is therefore that you **restrict** as much as possible the usage of the “**INNER JOIN**” operator, using it **only if it is an absolute must** to suppress all non-matching records. Otherwise, use an **Outer-Join** operator.

#### K.4.9 Why should I avoid using “SELECT \*”?

Because it may create problems in field formatting, field ordering, field names, and when making database modifications.

It may seem convenient to use “**SELECT \***” because whenever possible you avoid writing all the **output** field names. In spite of being convenient, my advice is you **never use** “**SELECT \***” as the **outermost Select** operation of a Query **nor** as an **input** record-list to a **Union operation**. As a general rule, my advice is that you avoid using use “**SELECT \***”.

I briefly explain some of the problems that you may face if you use it.

If you use “**SELECT \***” as the **outermost Select** operation of a Query MS-Access **will not show** the Query fields in “**Design View**”, and therefore, you cannot configure **the format** in which you want MS-Access to present each field of the Query results in “**Datasheet View**”.

Is you use “**SELECT \***” as an **input** record-list to a **Union** operation, it is easy to make mistakes with the field order because of the possible **differences of field order** in “**Datasheet View**”, “**Design View**” and “**SQL View**”. In case you **really** want to use asterisk, I will explain what is the field order it produces. The field order that “**SELECT \***” produces over an **input** Table is the **field order of the Table** in “**Design View**”. The field order that “**SELECT \***” produces over an **input** Query is the **field order of the Query** in “**SQL View**” (and **not** the field order of the Query in “**Datasheet View**”, nor the one in “**Design View**”).

If you use “**SELECT \***”, its **output field order** is **the same** as its **input field order**. If at a given moment you modify the Query code, and you **change** the **order** of records in some **inner** operation of the Query, this change **propagates** to **outer** SQL operations.

Since the **Union** operations are **order-dependent**, a change in the field order may cause them to malfunction. This problem does not happen if you use **Select** with an explicit list of fields, because its **output field order** is totally independent of its **input field order**.

If you use “**SELECT \***” you may have problems when you do modifications to your database design. If you want more detail on this, you may click *I.4*, *I.5* and *I.6*.

#### **K.4.10 Why should I avoid using the same name of an input field name for an output expression?**

Because this will force you to qualify the usage of all such duplicated **input** field names, and in case you forget to qualify it once, it will cause an error. In the following example, the SQL code of “**Inner\_select**” contains an **output** field named “**Age**”. The outer **Select** wants to present “**Age**” to the square and “**Age**” plus 3. The SQL code<sup>115</sup> would be:

```
SELECT (Inner_select.Age^2) AS Age, Age+3 AS Age_plus_3
FROM
  (SELECT Num AS Age FROM T_Numbers) AS Inner_select
```

However, the value of “**Age\_plus\_3**” is **not** the value of the **input** field name “**Age**” plus 3. Rather, it is the value of the **output** field name **Age** (i.e., the **input** field **Age** to the square) plus 3.

In cases where you want to use a **similar name** for an **output** field as the **name** of an **input** field, my advice is to use as **output** field name the **input** field name with the “**\_**” suffix. This can be alternated as a series of Queries (chains of unions, ...) to distinguish the **input** field names and the **output** field names in every Query. This is clearer, and you avoid having to qualify the **input** field name. The resulting code would be:

```
SELECT Age^2 AS Age_, Age+3 AS Age_plus_3
FROM
  (SELECT Num AS Age FROM T_Numbers) AS Inner_select
```

The Table “**T\_Numbers**” used in the example above is an auxiliary Table with only one field (named “**Num**”) that just contains integer numbers (click *K.2.2*).

#### **K.4.11 When are “SELECT” expressions evaluated along Query processing?**

You may **believe** that each “**SELECT**” **expression** is evaluated when computing the SQL operation where it belongs, and its **result** is **passed-on** to its enclosing SQL operation.

However, this is **not correct**! Each “**SELECT**” **expression** is **not** evaluated in each SQL operation and rather the expression itself is **passed-on as such** to its enclosing SQL operation. Each SQL operation builds **larger expressions** using, **as such**, the “**SELECT**” **expressions received** from its **enclosed** SQL operation(s) and passes, **as such**, the **larger expressions** it has built to its **enclosing** SQL operation.

An expression is **only** evaluated<sup>116</sup> when it is required in an SQL clause (e.g., a “**WHERE**”,

<sup>115</sup> This example and the next one are included in the Query “**J\_Field\_names**” from file “**Company\_Database.accdb**”.

<sup>116</sup> The way the database engine works is actually more complex than this, but this explanation is a

“**HAVING**” or “**ON**” clause) and in the **outermost** “**SELECT**” **expressions** of the Query that **you are running**. Notice that if a Query invokes auxiliary Queries, the expressions will **not** be evaluated in the outermost “**SELECT**” **expressions** of the **auxiliary Queries** and will **only** be evaluated in the outermost “**SELECT**” **expressions** of **the** Query you are running.

This implies that it is a **very bad practice** to handle **Nulls** in some enclosing SQL operation. It is much better practice to handle **Nulls** right in the SQL operation that has a database Table as an **input** record-list (click K.5).

## **K.5 Why and how should I carefully handle Nulls in my Queries?**

Because otherwise you will get a Query **crash** or an **exception-value**. For more information you may click:

- “K.5.1 What is a Null?”
- “K.5.2 What problems can Null produce?”
- “K.5.3 How is a Null produced?”
- “K.5.4 How do I handle Nulls in my Queries?”
- “K.5.5 Where do I handle Nulls in my Queries?”
- “K.5.6 Why is MS-Access not handling Null fields as I indicated?”

### **K.5.1 What is a Null?**

If at this moment you are reading this **Lightning Guide linearly**, click to read “C.6 What is a Null?” and then return here (you return simultaneously pressing the “**Alt**” and “**←**” keys).

### **K.5.2 What problems can Null produce?**

If at this moment you are reading this **Lightning Guide linearly**, click to read “J.14 What Null-related bugs can I get?” and then return here (you return by simultaneously pressing the “**Alt**” and “**←**” keys).

### **K.5.3 How is a Null produced?**

**Nulls** can appear in:

- **Fields** from **Tables** and from **auxiliary Queries**.
- **Results** from **functions** and **operators**.

#### **Nulls in fields from Tables and from auxiliary Queries**

- **Nulls** can **exist** in any **Table field** configured as “**Required=No**”.
- Remind that **Nulls** can even **exist** in a Table field configured as “**Required=Yes**” if they were already in the Table **before** the field had that configuration. I strongly advice that you configure almost **all** your Table fields as “**Required=Yes**” and to make

---

reasonable compromise between clarity and correctness.



sure that they do not contain pre-existing **Nulls**. Doing this will both prevent **Nulls**, and allow the Table field to belong to **Key** fields and candidate **Key** indexes.

- **Nulls should not exist** in any **field** returned by an **auxiliary Query**. They should not exist because you should program your Queries not to return **Null**. However, it is possibly that an auxiliary Query returns **Nulls**, so you should be aware of this.
- **Nulls are created in pointers to fields from Tables** by the “**LEFT JOIN**” and “**RIGHT JOIN**” operators.  
The SQL “**LEFT JOIN**” operator **creates Null** in **all** the **pointers to used fields** from **all** Tables **contained** in its **right (second)** SQL operation, including those in auxiliary Queries (**recursively!**). These **Nulls** are **created** in **all** the **pointers** to Table **fields** used in the “**SELECT**” **expressions** of the **left** input records that **did not match** the “**ON**” **Boolean** expression.  
The SQL “**RIGHT JOIN**” operator does the same but exchanging the **left (first)** input SQL operation and fields with the **right (second)** one.

#### Nulls in results from functions and operators

- All **aggregate** functions, except “**Count (\*)**”, “**DCount('\*')**”, “**Count ()**” and “**DCount()**”, **create** a returned **Null** if **all** the records in its input record-list produce **Null** in the argument expression, or, if the input record-list is empty. Just for completeness, notice that the four functions “**Count (\*)**”, “**DCount('\*')**”, “**Count ()**” and “**DCount()**” return the valid value “**0**” in that case.
- The aggregate functions “**StDev ()**”, “**DStDev()**”, “**StDevP ()**”, “**DStDevP()**”, “**Var ()**”, “**DVar()**”, “**VarP ()**” and “**DVarP()**” **create** a returned **Null** if their input record-list contains **zero or one** record(s) that produce a **non-Null** value in their argument expression.
- User-defined VBA functions may **create** a returned **Null** even if none of its arguments is **Null**.
- Value operators and most built-in VBA functions **will not create** a returned **Null**. However, most of them will return **Null** if one (or more) of their operands/arguments is **Null**.

#### K.5.4 How do I handle Nulls in my Queries?

You usually handle them differently depending on the expression:

- **Boolean expressions**  
These are the “**WHERE**”, “**ON**” and “**HAVING**” expressions.
- **Non-Boolean expressions**  
These are the “**TRANSFORM**”, “**SELECT**”, “**GROUP BY**”, “**PIVOT**” and “**ORDER BY**” expressions.

#### Handling Nulls in *Boolean* expressions

These are the “**WHERE**”, “**ON**” and “**HAVING**” expressions.

You usually handle **existing** or **created** **Nulls** by carefully designing your **Boolean**

expressions. There are three main reasons for this:

- The *Boolean* operators (click *G.5.5*) neither **crash** nor produce an **exception-value** when having **Null** argument(s). They just produce the corresponding result, which is the result you would expect interpreting **Null** as “**unknown**”.
- The SQL clauses using *Boolean* expressions work well (i.e., do not crash or produce an exception-value) if the *Boolean* expression returns **Null**: they will just interpret the **Null** as a **no-match**, this is, the same as *False*.
- The *Boolean* expressions from these clauses are **not** used in **any other** expression, unlike it happens with other ones (e.g., “**GROUP BY**” expressions are used in “**SELECT**” expressions).

If you wanted to handle a **field** or **function/operator result** called “**Suspect\_Null**” that can be **Null**, you would just write a suitable *Boolean* expression. Some examples of suitable *Boolean* expression could be the following:

```
ON Suspect_Null AND Right_op.Right_field
WHERE Nz(Suspect_Null, True) XOR Some_field
HAVING Iif(IsNull(Suspect_Null), Field_A AND Field_B, Field_C XOR Field D)
```

### Handling Nulls in Non-Boolean expressions

These are the “**TRANSFORM**”, “**SELECT**”, “**GROUP BY**”, “**PIVOT**” and “**ORDER BY**” expressions.

You usually **handle existing** or **created** Nulls using functions that check if a **field** or **function/operator result** is **Null** and in case it is **Null** they return a suitable valid value. If you wanted to handle a **field** or **function/operator result** called “**Suspect\_Null**” that can be **Null**, you would typically **enclose** it in an “**Iif()**” function as follows:

```
Iif(IsNull(Suspect_Null), Valid_value, Suspect_Null), ...
```

where “**Valid\_value**” represents a **valid** value that can be correctly handled in other expressions that can later evaluate this field.

As indicated above, I use mainly “**Iif()**” combined with “**IsNull()**” to handle “**Nulls**”, and I use very seldom “**Nz()**”, “**Switch()**” and “**Choose()**”. I do this because “**Nz()**”, “**Switch()**” and “**Choose()**” (see below) evaluate **all** the expressions in their arguments, regardless of the returned value. This makes these functions **less efficient** and also **more vulnerable to a crash** in case one of the non-returned expression produces a fatal error. Also, “**Nz()**” **always** returns a *String* when used in SQL, and this has some drawbacks. Finally, I prefer to use the “**IsNull()**” function instead of the “**IS NULL**” operator because the operand of “**IS NULL**” goes before the operator, and on my view this has worse readability.

To handle **Nulls**, you can also use the use the very useful built-in functions “**Switch()**” and “**Choose()**” (although the safest way is using “**Iif()**” with “**IsNull()**”). You can find a brief description of each of these functions clicking *G.6.1*.

Remind that even if **none** of your Table fields contains any **Null**, Query processing will likely **create Nulls**. If you want to know how a **Null** is created in your Queries, you may click “*K.5.3 How is a Null produced?*”.

### K.5.5 Where do I handle Nulls in my Queries?

You **must handle existing** or **created** Nulls at **the very first SQL operation where they appear**. If you handle them instead in an **enclosing** SQL operation, you risk that the **Null** is evaluated in an **intermediate** expression, causing an **exception-value** or a **Query crash**. Depending on the cause of a **Null**, you handle it in the following way:

- A **created** Null from a **function** result:  
You must handle the **Null** at **each and every expression** where the **function** is used.
- An **existing** or **created** Null from a **Table field**  
You must handle the **field** at **each and every expression** in the SQL operation (even if they are inside auxiliary Queries) where the **Table field** is used as an **input** record-list. Notice that you **must** individually handle **each and every** usage of the **Table field** in **all** the expressions (“**ON**”, “**TRANSFORM**”, “**SELECT**”, “**WHERE**”, “**GROUP BY**”, “**HAVING**”, “**PIVOT**” and “**ORDER BY**” expressions) of the corresponding SQL operation.

Let me add a little more detail for the case of **Nulls created** by the “**LEFT JOIN**” (or “**RIGHT JOIN**”) operators in **pointers** to **fields** of Tables:

- If the **right** (or **left**, respectively) **input** record-list is a Table name, then you **must** handle **each and every** Table **field** that is **used** in the “**ON**” expression, **and also**, in **all** the expressions in **all** the clauses of its **enclosing Select** operation.
- If the **right** (or **left**, respectively) **input** record-list is an SQL operation, then you **must** handle **each and every** Table **field** that is used in **each and every** expression from **each and every enclosed** SQL operation that has a **Table** as its **input** record-list. In case the SQL operation is itself an auxiliary Query name, or it contains auxiliary Query names, then you **must** handle **each and every** Table **field** that is used in **each and every** expression from **each and every** SQL operation from **each and every** auxiliary Query that has a **Table** as its **input** record-list. In case the auxiliary Queries have themselves other auxiliary Queries, you have to **recursively** apply this.

If you want more detail on how to properly handle **created** Nulls from “**LEFT JOIN**” and “**RIGHT JOIN**”, you may click “[K.5.6 Why is MS-Access not handling Null fields as I indicated?](#)”.

### K.5.6 Why is MS-Access not handling Null fields as I indicated?

The most frequent causes of this are:

- You are handling “**LEFT JOIN**” (or “**RIGHT JOIN**”) **Nulls** as if their **output** fields become **Null**, when it is rather **all** the **pointers** to **used fields** from **all** Tables **contained** in their **right** (or **left**, respectively) SQL expression what become **Null**. If you want to know more about this, you may click:
  - “[K.5.6.1 How are Null fields created in “LEFT JOIN” and “RIGHT JOIN”?](#)”
- You are **not** handling “**LEFT JOIN**” and “**RIGHT JOIN**” **Nulls** in each and every SQL operation that has a **Table** name as its **input** record-list (doing this recursively inside all auxiliary Queries). If you want to know more about this, you may click:

- “K.5.6.2 Why should I handle Null fields in all the innermost SQL operations?”

### K.5.6.1 How are Null fields created in “LEFT JOIN” and “RIGHT JOIN”?

When you handle **Nulls** produced by “**LEFT JOIN**” and “**RIGHT JOIN**”, you may think that the operator produces a **Null** in its corresponding **output** field, and therefore you handle the **Null** enclosing the **output** field in an “**Iff()**” function (you may click “K.5.4 How do I handle Nulls in my Queries?”).

However, this assumption is **not correct!** “**LEFT JOIN**” and “**RIGHT JOIN**” **do not** produce **Null** in their **output** fields. What they do is produce **Null** in **all** the **pointers** to **used fields** from **all** Tables and auxiliary Queries **contained** in their **right** or **left** (respectively) SQL expression.

Let me show you the difference with an example over same two Tables that I used for the **Join** operators (click F.8). One Table is a list of house owners, and the other Table is a list of car owners. The field “**ID**” represents a **unique** identification number to avoid mistaking two different persons that have the same name. The example Tables are:

T_House_owners			T_Car_owners		
ID	Name	Address	ID	Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.	18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

Now let us try the following Query<sup>117</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
(
  SELECT ID, Name, Address, 1 AS Null_expected
  FROM T_House_Owners
) AS Houses
RIGHT JOIN
  T_Car_Owners          AS Cars
ON Houses.ID = Cars.ID
WHERE IsNull(Null_expected)
```

If the “**RIGHT JOIN**” produced **Null** in all the **left** fields of **non-matching right** records, this SQL operation would produce the **non-matching** records from the **right** record-list, because I have used the “**WHERE**” expression:

```
IsNull(Null_expected)
```

This “**WHERE**” expression should **remove all** the **matching** records (because in these records the field “**Null\_expected**” is expected to be **1**) and it should **retain all** the **non-matching right** records (because in these records the field “**Null\_expected**” is expected to be **Null**). However, the **output** record-list from the previous SQL operation

<sup>117</sup> This is the Query “J\_Nulls\_constant\_1” from file “Company\_Database.accdb”

is **empty**: there is **no record** that satisfies the “**WHERE**” expression!

To check what is happening, let us modify the Query above by removing the “**WHERE**” clause, and adding the “**SELECT**” expression “**Null\_expected**”, to see what is going on<sup>118</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Null_expected
      , Cars.ID, Cars.Name, Car
FROM
  (
    SELECT ID, Name, Address, 1 AS Null_expected
    FROM T_House_Owners
  ) AS Houses
RIGHT JOIN
  T_Car_Owners          AS Cars
ON Houses.ID = Cars.ID
```

The **output** from this Query is:

J_Nulls_constant_1					
Houses.ID	Houses.Name	Address	Null_expected	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	1	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	1	Xi Liu	Ford Focus
			1	John Welsh	Opel Zafira
			1	Juan Perez	Ford Mustang

You may see that **all** the records from this “**RIGHT JOIN**”, even the non-matching ones (the last two), have the **left** field “**Null\_expected**” with value “**1**”.

The reason is precisely what I indicated at the beginning of his section: “**LEFT JOIN**” and “**RIGHT JOIN**” **do not** produce **Null** in their **output** fields. What they do is produce **Null** in the pointers to **Table fields** used in expressions. In this example, there are **no Table fields** used to compute “**Null\_expected**” and it is just the **constant** value “**1**”. Therefore, the “**LEFT JOIN**” produces in the field “**Null\_expected**” the constant value “**1**” and **does not** produce **Null**.

If you want to know how to write an SQL operation that produces all the **non-matching left** or **right** records of an **Join**, you may click “*K.6.7 How do I produce the non-matching records of a Join operation?*”.

### K.5.6.2 Why should I handle Null fields in all the innermost SQL operations?

Because “**LEFT JOIN**” and “**RIGHT JOIN**” unhandled **Null** in pointers to **Table fields** will likely produce a Query **crash** or an **exception-value** (click *J.15.1*) when evaluated in an expression from an enclosing SQL operation. If an exception-value has been produced, it **cannot be handled** (click “*J.15.3 How do I prevent exception-values?*”) using functions such as “**Iif()**”, “**IsNumeric()**” or “**IsError()**”. The **exception-value** will then either **crash** the Query or be **shown** in the Query results.

You **must** therefore handle **Null** pointers to **Table fields** in **each and every** “innermost” SQL operation. By “innermost” SQL operation I mean an SQL operation that has a **Table**

<sup>118</sup> This is the Query “**J\_Nulls\_constant\_2**” from file “**Company\_Database.accdb**”

as its **input** record-list. In case there are auxiliary Queries, then you have to handle **Null fields in each and every** “innermost” SQL operation, **recursively** across **all** auxiliary Queries.

Let me show you this with an example over the Table “**T\_Capital\_Rainfall**” from *F.10*. The following **two Select** operations<sup>119</sup> work well, and do not produce **Null**:

```
SELECT Capital
FROM T_Capital_Rainfall

SELECT Capital, Cdbl(Cal_Year) AS Cdbl_Year
FROM T_Capital_Rainfall
WHERE Capital = "Beijing"
```

You now build the following “**LEFT JOIN**” with both of them, using the “**Iif()**” and “**IsNull()**” functions to handle the **Nulls** created in the Table **fields** inside the **right** SQL operation (in the **non-matching** records)<sup>120</sup>:

```
SELECT Left_op.Capital
, Iif(IsNull(Cdbl_Year), 1900, Cdbl_Year) AS Correct_Year
FROM
(
    SELECT Capital
    FROM T_Capital_Rainfall
) AS Left_op
LEFT JOIN
(
    SELECT Capital, Cdbl(Cal_Year) AS Cdbl_Year
    FROM T_Capital_Rainfall
    WHERE Capital = "Beijing"
) AS Right_op
ON Left_op.Capital = Right_op.Capital
```

You probably expect this Query to show the year from Table “**T\_Capital\_Rainfall**” for the case of Beijing, and the year “**1900**” for all **other** capital cities in “**T\_Capital\_Rainfall**”. If you try it, you will see that this is **not** the result. The result is the year from “**T\_Capital\_Rainfall**” for the case of Beijing, and the exception-value “**#Error**” for all **other** capital cities in “**T\_Capital\_Rainfall**”.

The reason is that the **pointers** to Table fields “**Capital**” and “**Cal\_year**” become **Null** in the right SQL operation. This is, the outermost “**SELECT**” **expressions** are **evaluated**

---

<sup>119</sup> These are the Queries “**J\_Nulls\_SQL\_Operators\_1**” and “**J\_Nulls\_SQL\_Operators\_2**” from file “**Company\_Database.accdb**”.

<sup>120</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_3**” from file “**Company\_Database.accdb**”.



(click *K.4.11*) as if the SQL operation was written in the following way<sup>121</sup>:

```
SELECT Left_op.Capital
      , Iif(IsNull(Cdbl(Cal_Year)), 1900, Cdbl(Cal_Year)) AS Correct_Year
FROM
  (
    SELECT Capital
    FROM T_Capital_Rainfall
  ) AS Left_op
LEFT JOIN
  (
    SELECT Capital, Cal_Year
    FROM T_Capital_Rainfall
    WHERE Capital = "Beijing"
  ) AS Right_op
ON Left_op.Capital = Right_op.Capital
```

As you may see in this example, MS-Access **replaces** the “**SELECT**” **expression names** (what I have called “**Output-field-names**” in other parts of this book) along your SQL operations by the **actual expressions corresponding** to the **expression names**. This produces **expanded “SELECT” expressions** in the **outermost** SQL operation (and in intermediate clauses along the way) that use the **raw Table field names** from the **innermost** SQL operations that **do not** contain any **expression name**. In this example, the resulting outermost “**SELECT**” **expression** is:

```
Iif(IsNull(Cdbl(Cal_Year)), 1900, Cdbl(Cal_Year)) AS Correct_Year
```

You may be thinking “so what”. The problem is that the “**LEFT JOIN**” operator will create **Nulls** in the **Table field** “**Cal\_Year**” (click *K.5.6.1*) from the **right** SQL operation. Therefore, the expression above **does not** handle these **Nulls** properly, and rather the expression produces the exception-value “**#Error**”. This happens because the “**IsNull()**” function is applied over “**Cdbl(Cal\_Year)**” and not over “**Cal\_Year**”, and “**Cdbl(Cal\_Year)**” returns “**#Error**”: once the exception-value has been produced, it **cannot** be handled, and the Query produces “**#Error**” instead of the desired value “**1900**”.

You **fix** this by adjusting your Query, so the **Null** pointer to the Table field is handled **right when it appears**, in the **left** input record-list of the “**LEFT JOIN**”, and **not** in its **output** record-list. You can code this in the following way<sup>122</sup>:

```
SELECT Left_op.Capital, Cdbl_Year AS Correct_Year
FROM
  (
    SELECT Capital
    FROM T_Capital_Rainfall
  ) AS Left_op
LEFT JOIN
  (
    SELECT Capital
      , Cdbl(Iif(IsNull(Cal_Year), 1900, Cal_Year)) AS Cdbl_Year
    FROM T_Capital_Rainfall
    WHERE Capital = "Beijing"
  ) AS Right_op
ON Left_op.Capital = Right_op.Capital
```

<sup>121</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_4**” from file “**Company\_Database.accdb**”.

<sup>122</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_5**” from file “**Company\_Database.accdb**”.

This last Query produces the desired result: the year in the Table for Beijing and “1900” for all other capital cities.

Notice that the field “**Capital**” is used in the “**WHERE**” and in the “**ON**” *Boolean* expressions, and it is handled just by writing the *Boolean* expressions that produce the desired result. As you may see, in SQL clauses with *Boolean* expressions it is **not required** to handle **Null** fields using the “**Iif()**” and “**IsNull()**” functions.

Let me show you another example<sup>123</sup> with additional nested SQL expressions:

```
SELECT Left_op.Capital, Cdbl_Year AS Correct_Year
FROM
  T_Capital_Rainfall AS Left_op
LEFT JOIN
  (
    SELECT Capital
    , Cdbl(Iif(IsNull(CInt_Year, 1900, CInt_Year)) AS Cdbl_Year
    FROM
      (
        SELECT Capital, CInt_Year
        FROM
          (SELECT Capital, CInt(Cal_Year) AS CInt_Year
          FROM T_Capital_Rainfall )
        )
    WHERE Capital = "Beijing"
    ) AS Right_op
ON Left_op.Capital = Right_op.Capital
```

Notice that I am handling the **Null** field “**Capital**” in the **outermost right Select operation**, just like I did in the previous example. The problem is that in this current example the **outermost right Select operation** is **not** the one having the Table as its **input** record-list, and also, that there is an intermediate expression “**CInt(Cal\_Year)**” that produces an exception-value when processing the unhandled **Null**.

You might think that a solution could be enclosing the two innermost Select operations in an **auxiliary Query**, and the auxiliary Query would behave as a Table. This is, you would first create the following auxiliary Query<sup>124</sup> with the name “**J\_Nulls\_SQL\_Operators\_7**”:

```
SELECT Capital, CInt_Year
FROM
  (SELECT Capital, CInt(Cal_Year) AS CInt_Year
  FROM T_Capital_Rainfall )
```

<sup>123</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_6**” from file “**Company\_Database.accdb**”.

<sup>124</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_7**” from file “**Company\_Database.accdb**”.



and then you would use it by modifying the main Query as follows<sup>125</sup>:

```
SELECT Left_op.Capital, Cdbl_Year AS Correct_Year
FROM
    T_Capital_Rainfall AS Left_op
LEFT JOIN
    (
        SELECT Capital
            , Cdbl(Iif(IsNull(CInt_Year), 1900, CInt_Year)) AS Cdbl_Year
        FROM J_Nulls_SQL_Operators_7
        WHERE Capital = "Beijing"
    ) AS Right_op
ON Left_op.Capital = Right_op.Capital
```

However, **this does not work!** The reason is that an auxiliary Query **does not** behave as a Table, and **all** the **pointers** to actual Table fields **inside** the auxiliary Query become **Null!**

You therefore have to handle the **Null pointers** to Table **fields** in **all** the SQL operations that have a Table name as its **input** record-list. In this case, this means handling the **Null** fields in the **Select** operation that has the Table “**T\_Capital\_Rainfall**” as its **input** record-list. This can be done with the following SQL code<sup>126</sup>, that is now correct:

```
SELECT Left_op.Capital, Cdbl_Year AS Correct_Year
FROM
    T_Capital_Rainfall AS Left_op
LEFT JOIN
    (
        SELECT Capital
            , Cdbl(CInt_Year) AS Cdbl_Year
        FROM
            (
                SELECT Capital, CInt_Year
                FROM
                    (SELECT Capital, CInt(Iif(IsNull(Cal_Year), 1900, Cal_Year))
                        AS CInt_Year
                    FROM T_Capital_Rainfall
                )
            )
        WHERE Capital = "Beijing"
    ) AS Right_op
ON Left_op.Capital = Right_op.Capital
```

Notice that the field “**Capital**” is used in the “**WHERE**” and in the “**ON**” **Boolean** expressions, and it is handled just by writing the **Boolean** expressions that produce the desired result. As you may see, in SQL clauses with **Boolean** expressions it is **not required** to handle **Null** fields using the “**Iif()**” and “**IsNull()**” functions.

## **K.6 What are some useful models of SQL code?**

You may click:

- “*K.6.1 How do I generate/create record-lists?*”
- “*K.6.2 How do I replicate record-lists?*”
- “*K.6.3 How do I produce totals in addition to individual results?*”

<sup>125</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_8**” from file “**Company\_Database.accdb**”.

<sup>126</sup> This is the Query “**J\_Nulls\_SQL\_Operators\_9**” from file “**Company\_Database.accdb**”.

- “K.6.4 How do I convert rows into columns?”
- “K.6.5 How do I convert columns into rows?”
- “K.6.6 How do I use a Transform operation to show a cross table?”
- “K.6.7 How do I produce the non-matching records of a Join operation?”
- “K.6.8 How do I write a Full-Outer-Join?”
- “K.6.9 How do I “merge” two record-lists?”
- “K.6.10 How do I produce a weighted average?”
- “K.6.11 How do I design a data check Query?”
- “K.6.12 How do I get the exact record ordering I want?”
- “K.6.13 How do I use String fields to display different data types in the same Query column?”
- “K.6.14 What are useful tricks with SQL aggregate functions?”

### **K.6.1 How do I generate/create record-lists?**

A **Select** operation over an **auxiliary Table of numbers** generates record-lists.

For example, the following SQL operation<sup>127</sup>:

```
SELECT Date()+Num AS Next_30_Days
FROM T_Numbers
WHERE Num BETWEEN 1 AND 30
```

generates a record-list with **30 records**, each of them containing a date between tomorrow and tomorrow plus 30 days.

Another example is the following SQL operation<sup>128</sup> that produces a record-list where each record has a date in the next month:

```
SELECT DATESERIAL(YEAR(Date()), Month(Date())+1, Num) AS Days_Next_Month
FROM T_Numbers
WHERE (Num BETWEEN 1 AND 31)
AND ( Month(DATESERIAL(YEAR(Date()), Month(Date())+1, Num))
= (Month(Date())+1) )
```

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click K.2.2).

As you can see from the examples above, using more complex expression in the output-fields, and more complex expressions in the “**WHERE**” clause, will allow you to flexibly generate the record-lists that you need in a simple and efficient way.

You should also know that SQL **allows** you to **create** record-lists without the need of any Table, but this can only produce **one** output record, and it must be the **only** SQL operation of the Query. The following is an example<sup>129</sup>:

```
SELECT "New York" AS City, "Manhattan" AS District ;
```

<sup>127</sup> This is the Query “K\_Next\_30\_dates” from file “Company\_Database.accdb”.

<sup>128</sup> This is the Query “K\_Dates\_next\_month” from file “Company\_Database.accdb”.

<sup>129</sup> This is the Query “F\_Select\_without\_FROM” from file “Company\_Database.accdb”.

this SQL Query code creates **one output** record with fields “City” and “District”, and the values indicated in each field. Since you can only generate **one** record, and you **cannot** combine the SQL operation above with other similar ones using the **Union** operator, this SQL operation is only useful to do tests and in very few other particular cases.

### K.6.2 How do I replicate record-lists?

A **Cross-Join** (click [F.8.2](#) and [F.8.5](#)) between a given **input** record-list and a record-list with “n” records **replicates** the **input** record-list “n” times.

For example, the following SQL operation:

```
(
  SELECT Num
  FROM T_Numbers
  WHERE Num BETWEEN 1 AND 2
) AS Replicator
, -- The comma in this line is the Cross-Join operator
  T_Capital_Rainfall_Q AS Records_to_Replicate
```

**duplicates** the record-list from the Table “T\_Capital\_Rainfall\_Q” (see at the end of this section) and adds the field “Num” to distinguish the records belonging to each of the two copies. One of the input record-list replicas has field “Num” with value “1” and the other one with value “2”. If you wanted more copies, it is as simple as replacing the “2” in the SQL operation above **by the number of copies** that you want.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

One useful application of replicating record-lists is producing partial and total results in addition to the individual results. For example, if you want to show the rainfall data by quarters, by years and by periods of five consecutive years. If you want to know more about this, you may click “[K.6.3 How do I produce totals in addition to individual results?](#)”.

You can also replicate record-lists with a **Union** operation. However, if you do it with the **Union** operation you need to copy the **whole** SQL code of the operation whose records want to replicate **as many times** as you want to replicate it, and **also**, you have to **edit** each copy in case you want a differentiator field to distinguish each copy. This will make your Query code longer, more difficult to read and more difficult to write. My advice is you always use a **Cross-Join** as the one in the example above when you want to replicate record-lists.

The Table “T\_Capital\_Rainfall\_Q” used in the example above is the one of quarterly rainfall measurements in capital cities used in several other examples along this Lightning Guide:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

### K.6.3 How do I produce totals in addition to individual results?

If you just want **one** aggregate **total** row, you may click “H.8 How do I show aggregate values (e.g., totals) in a Table/Query/Form?”.

Otherwise, a **Select-group by aggreg** (click F.7.2 and F.7.6.2) over a **duplicated** record-list (click the previous section K.6.2) produces **totals** in addition to **individual** results.

Let me show you an example<sup>130</sup> over the Table “T\_Capital\_Rainfall\_Q” that I just showed at the end of the previous section K.6.2:

```
SELECT Capital AS Cap_City, Cal_Year AS C_Year
      , Iif(Num=1, "TOTAL", Quart) AS Quarter
      , Sum(Quart_Rainfall) AS Rainfall
FROM
  (
    SELECT Num
    FROM T_Numbers
    WHERE Num BETWEEN 1 AND 2
  ) AS Duplicator
, -- The comma in this line is the Cross-Join operator
  T_Capital_Rainfall_Q AS Records_to_Duplicate
GROUP BY Capital, Cal_Year, Iif(Num=1, "TOTAL", Quart) ;
```

this SQL operation above produces each capital city with its **quarterly** rainfall data (individual results), and **also**, its yearly **totals**. Its **output** record-list will be:

K_Partials_and_Totals			
Cap_City	C_Year	Quarter	Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Beijing	2018	TOTAL	28.8
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7
Washington	2018	TOTAL	32.76

<sup>130</sup> This is the Query “K\_Partials\_and\_Totals” from file “Company\_Database.accdb”.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click K.2.2).

The guidelines to produce **totals in addition to individual results** in a general case are the following:

1. You do a **Cross-Join** between the **input** record-list and a record-list having **two** records and having a **field** (that I will call “Num”) with **distinct** values. This record-list that has a fixed number of distinct records you can copy from the example above (click K.6.1 for an explanation).
2. You enclose the **Cross-Join** from point 1 in a **Select-group\_by\_aggreg** using as “**GROUP BY**” **expressions** the **input fields**, such that for **each distinct joint combination of their values** you get **one** “**TOTAL**” record.
3. You add one additional “**GROUP BY**” **expression** consisting of an “**Iif()**” function. The first argument of the “**Iif()**” function is “**Num=1**”, the second argument is the string “**"TOTAL"**” and the third argument is the **input** field that **identifies** each **individual** value.
4. The “**SELECT**” **expressions** are the “**GROUP BY**” **expressions** plus the “**Sum()**” SQL aggregate function over the **input** field that **contains** each **individual** value.

Let us check how I followed these guidelines in the example above:

1. I did a **Cross-Join** between the **input** record-list “**T\_Capital\_Rainfall\_Q**” and a record-list having **two** records and having a **field** (called “**Num**”) with **distinct** values.
2. I enclosed the **Cross-Join** from point 1 in a **Select-group\_by\_aggreg** using as “**GROUP BY**” **expressions** the **input fields** such that for **each distinct joint combination of their values** you get **one** “**TOTAL**” record. In the example these fields are “**Capital**” and “**Cal\_Year**”.
3. I added one additional “**GROUP BY**” **expression** consisting of an “**Iif()**” function. The first argument of the “**Iif()**” function is “**Num=1**”, the second argument is the string “**"TOTAL"**” and the third argument is the **input** field that **identifies** each **individual** value, that in this case is “**Quart**”.
4. The “**SELECT**” **expressions** are the “**GROUP BY**” **expressions** plus the “**Sum()**” SQL aggregate function over the **input** field that **contains** each **individual** value, that in this case is “**Quart\_Rainfall**”.

Notice that **totals** can be produced at **several** levels. You could have for example rainfall **by Quarter**, the total **by calendar year**, and the total **by period of 10 years**. To do this you would need to **replicate three times** the record-list (click K.6.1) and change the “**Iif()**” expression in the “**GROUP BY**” and “**SELECT**” clauses. Notice further that instead of doing the **sum**, you may want the **average**, or the **maximum**, or the **minimum**. This is trivial to do just by using the corresponding SQL aggregate function “**Avg()**”, “**Max()**” or “**Min()**” respectively.

As I indicated at the beginning of this section, in case you want **only one total row** for

the whole Query results, then you do not need to code it in SQL because MS-Access has a **tool** to produce **one total row** for you. If you want to know how to produce this total row, you may click “*H.8 How do I show aggregate values (e.g., totals) in a Table/Query/Form?*”. Showing “**Datasheet View**” aggregate values using the **Total “Σ”** icon (click *H.8*) is very useful. However, on many cases you may want to produce **several** aggregate records (e.g., totals per Quarter), or selective aggregates (e.g., average only of the top branches). Also, you may want the aggregate records being part of the output record-list of a Query, instead of being a value shown by MS-Access in “**Datasheet View**”: for example, if you want the aggregate values of a Query to be further processed by other Queries.

#### K.6.4 How do I convert rows into columns?

A **Select-group\_by\_aggreg** (click *F.7.2* and *F.7.6.2*) converts rows into columns.

Let me show you an example<sup>131</sup> over the Table “T\_Capital\_Rainfall\_Q” (see at the end of this section):

```
SELECT Capital, Cal_Year
      , Max(Iif(Quart="Q1", Quart_Rainfall, Null)) AS Q1
      , Max(Iif(Quart="Q2", Quart_Rainfall, Null)) AS Q2
      , Max(Iif(Quart="Q3", Quart_Rainfall, Null)) AS Q3
      , Max(Iif(Quart="Q4", Quart_Rainfall, Null)) AS Q4
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year ;
```

this SQL operation above produces the following output record-list:

K_Rows_into_columns_1					
Capital	Cal_Year	Q1	Q2	Q3	Q4
Beijing	2018	0	4	7.8	17
Washington	2018	12.13	5.67	2.26	12.7

If you compare this output record-list with the Table “**T\_Capital\_Rainfall\_Q**” (copied below), you will see what I mean by “**convert rows into columns**”: it means producing **one** record for each **group** of four quarterly records, where the record contains the values of the **four** quarters. I am therefore turning **rows** (i.e., four **records**) into **columns** (i.e., the four **fields** in the new records with different structure).

The guidelines to **convert rows into columns** in a general case are the following:

1. You identify **one** “**PIVOT**” **expression** (over **input field names**) such that **each** of its **distinct values** will produce **one** “**PIVOT**” **column** (i.e., one “**PIVOT**” **field**). It is quite frequent to use **one input field name** as “**PIVOT**” **expression**.
2. You identify **several** “**GROUP BY**” **expressions** whose **distinct joint values** determine the **number** of **output** records, and also **uniquely identify each output** record. Each “**GROUP BY**” **expression** is built over the **input field names** (without using SQL aggregate functions). It is quite frequent to use as “**GROUP BY**” **expressions** some of the **input fields** that you want to **stay the**

<sup>131</sup> This is the Query “K\_Rows\_into\_columns\_1” from file “Company\_Database.accdb”.



same in the **output** records.

3. You identify one “**TRANSFORM**” **expression** built over the **input field names** (without using SQL aggregate functions). The “**TRANSFORM**” **expression** produces the **field values** of the “**PIVOT**” **fields** (i.e., “**PIVOT**” **columns**). It is quite frequent to use one **input field name** as “**TRANSFORM**” **expression**.
4. You do a **Select-group\_by\_aggreg** over the **input record-list** using the “**GROUP BY**” **expressions**. The “**SELECT**” **expressions** are the “**GROUP BY**” **expressions**, plus a number of “**other**” “**SELECT**” **expressions** that you want, plus as many “**Max ()**” functions as **distinct values** are produced by the “**PIVOT**” **expression**. It is quite frequent to use none “**other**” “**SELECT**” **expressions**.
5. The argument to each “**Max ()**” function is an “**Iif()**” function. The first argument of each “**Iif()**” function is an **equality** comparison between the “**PIVOT**” **expression** and one of the **distinct values** that it produces. The second argument is the “**TRANSFORM**” **expression** and the third argument is **Null**.
6. It is quite frequent to use as “**SELECT**” **field name** corresponding to each “**Max ()**” function the **value** (converted to *String*) in the comparison operator of the first argument of its “**Iif()**” function. This is, one of the **distinct values** produced by the “**PIVOT**” **expression**.

The reason why you should use the SQL aggregate function “**Max ()**” (or “**Min ()**”) is that the “**Iif()**” functions will produce only one (or none) **non-Null** “**TRANSFORM**” **expression value** in each **group** of records, and this is the **value** that you want to show. Notice that you should **not** use “**First ()**” (nor “**Last ()**”) because they return the value from an **unknown** record **even if it is Null**. If you used “**First ()**” (or “**Last ()**”) you would get a pseudo-random combination of meaningless **Null** values and meaningful **values** from the “**TRANSFORM**” expression. However, “**Max ()**” (and “**Min ()**”) always return a **non-Null** value (as long as there is one) which is precisely what you want.

Let us check how I followed the guidelines in the example above:

1. I identified (as a frequent case) one **input field name**, “**Quart**”, as the “**PIVOT**” **expression**. **Each** of its **distinct values** will produce **one** “**PIVOT**” **column** (i.e., one “**PIVOT**” **field**). Therefore, we will have **four** “**PIVOT**” **fields**, corresponding to the **four distinct values** “**Q1**”, “**Q2**”, “**Q3**”, and “**Q4**”.
2. I identified (as a frequent case) one **input field name** as **each** of the two “**GROUP BY**” **expressions**: “**Capital**” and “**Cal\_Year**”. The **two distinct joint values** of these fields determine the **number** of **output** records, and also **uniquely identify each output** record.
3. I identified (as a frequent case) one **input field name** as the “**TRANSFORM**” **expression**: “**Quart\_rainfall**”. This **field** will produce the **field values** of the “**PIVOT**” **fields** (i.e., “**PIVOT**” **columns**).
4. I did a **Select-group\_by\_aggreg** over the **input record-list** using the “**GROUP BY**” **expressions** “**Capital**” and “**Cal\_Year**”. The “**SELECT**” **expressions** are the “**GROUP BY**” **expressions** (there are no “**other**” “**SELECT**”

expressions), plus **four** “**Max()**” SQL aggregate functions (one for each of the **four distinct values** of the “**PIVOT**” expression “**Quart**”: “**Q1**”, “**Q2**”, “**Q3**”, and “**Q4**”).

5. The argument to each “**Max()**” function is an “**Iif()**” function. The first argument of the “**Iif()**” function is an **equality** comparison between the “**PIVOT**” expression “**Quart**” and **one** of the **four distinct values** that it produces (i.e., “**Q1**”, “**Q2**”, “**Q3**”, and “**Q4**”). The second argument is the “transform” expression “**Quart\_rainfall**” and the third argument is **Null**.
6. As a frequent case, I used as the **output field name** corresponding to each “**Max()**” function the **value** (converted to *String*) in the comparison operator of the first argument of its “**Iif()**” function. These **four** “**PIVOT**” **field names** are therefore “**Q1**”, “**Q2**”, “**Q3**”, and “**Q4**”.

You can also turn rows into columns using a **Transform** operation (click *K.6.6* and *F.10*). The following **Transform** operation<sup>132</sup> will produce exactly the same output record-list shown above:

```
TRANSFORM First(Quart_Rainfall) AS GenVals
SELECT Capital AS Cap_City, Cal_Year AS C_Year
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year
PIVOT Quart ;
```

Since there are **two** options (**Select-group\_by\_aggreg** and **Transform**), you may wonder “which one should I use to covert rows into columns?”. The answer is:

- If you **do not need** to do further processing of the record-list, use a **Transform** operation.
- Otherwise, use a **Select-group\_by\_aggreg**.

The reason is that a **Transform** operation works for a **variable** number of rows to be turned into columns, you **do not** need to know **in advance** the values that will characterize each column produced from rows, and it is easier to write. These are **big** advantages of using a **Transform** operation. The drawback is that the **Transform** operation **must** be the outermost SQL operation in the Query, and therefore, it is **not possible** to do any further processing of its output record-list.

The main disadvantages of using a **Select-group\_by\_aggreg** are that the **number of rows** you turn into columns is **totally fixed** (in the example above is fixed to **four**) in the SQL code, and also, that you need to **know in advance** the field values that identify each column, in order to **write** the corresponding “**Iif()**” expressions **in the SQL code**.

Notice that converting rows into columns is the **reciprocal** operation to the one in the **next** section *K.6.5*, that addresses **converting columns into rows**.

The Table “**T\_Capital\_Rainfall\_Q**” used in the example above is the one of quarterly

---

<sup>132</sup> This is the Query “**K\_Rows\_into\_columns\_2**” from file “**Company\_Database.accdb**”.



rainfall measurements in capital cities used in several other examples of this Guide:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

### K.6.5 How do I convert columns into rows?

Replicating the record-list (click *K.6.2*) as many times as rows you want converts columns into rows.

Let me show you the following example<sup>133</sup> over the results of the Query “K\_Rows\_into\_columns\_1” from the previous section *K.6.4*:

```
SELECT Capital AS Cap_City, Cal_Year AS C_Year
      , Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4,"Q4") AS Quart
      , Switch(Num=1, Q1, Num=2,Q2, Num=3,Q3, Num=4,Q4) AS Quart_rainfall
FROM
  (SELECT Num FROM T_Numbers WHERE Num BETWEEN 1 AND 4) AS Quadr
,
  K_Rows_into_columns_1
ORDER BY Capital, Switch(Num=1,"Q1", Num=2,"Q2", Num=3,"Q3", Num=4,"Q4")
```

this SQL operation above produces the same record-list as the one in the Table “T\_Capital\_Rainfall\_Q” (copied at the end of the previous section *K.6.4*).

If you compare the results of “K\_Rows\_into\_columns\_1” and the records of Table “T\_Capital\_Rainfall\_Q”, you will see what I mean by “**converting columns into rows**”: it means producing **four** records, each with **one** quarterly value, from **each record** that has **four** quarterly values. I am therefore turning **columns** (i.e., four **fields**) into **rows** (i.e., four **records**).

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click *K.2.2*).

The guidelines to **convert columns into rows** in a general case are the following:

1. You do a **Cross-Join** between the **input** record-list and a record-list having **as many records** as the **number of columns** that you want to convert into rows and having a **field** (that I will call “**Num**”) with **distinct** values. This record-list that has a fixed number of distinct records you can copy from the example above (click *K.6.2* for an explanation).

<sup>133</sup> This is the Query “K\_Columns\_into\_rows” from file “Company\_Database.accdb”.

2. You enclose the **Cross-Join** from point 1 in a **Select**. Its “**SELECT**” **expressions** are **each** of the “columns” (i.e., fields) that you **do not** want to convert into rows, plus **two** specific “**Switch()**” functions (click G.6).
3. **Both** “**Switch()**” functions from point 2 have the same **Boolean** expression arguments (the **odd** arguments). Each **odd** argument is an equality function between the “**Num**” field and **each** of the **distinct** values (constants) that it has in the record-list from point 1.
4. The **even** arguments of **one** of the “**Switch()**” functions are the **field names** of the “columns” that you want to convert into rows.
5. The **even** arguments of **the other** “**Switch()**” function are the **fields** (i.e., “columns”) that you want to convert into rows, **in the same order** as in the “**Switch()**” from point 4.

Let us check how I followed these guidelines in the example above:

1. I did a **Cross-Join** between the **input** record-list (the Query “**K\_rows\_into\_columns\_1**”) and a record-list having 4 records, because I want to convert 4 columns into rows. This second record-list has a field “**Num**” with **distinct** values.
2. I enclosed the **Cross-Join** from point 1 in a **Select**. Its “**SELECT**” **expressions** are **each** of the “columns” (i.e., fields) that I **do not** want to convert into rows (“**Capital**” and “**Cal\_Year**”) plus **two** specific “**Switch()**” functions (click G.6).
3. **Both** “**Switch()**” functions from point 2 have the same **Boolean** expression arguments (the **odd** arguments). Each **odd** argument is an equality function between the “**Num**” field and **each** of the **distinct** values (constants) that it has in the record-list from point 1: “**Num=1**”, “**Num=2**”, “**Num=3**” and “**Num=4**”.
4. The **even** arguments of **one** of the “**Switch()**” functions are the **field names** of the “columns” that I wanted to convert into rows: “**Q1**”, “**Q2**”, “**Q3**” and “**Q4**”.
5. The **even** arguments of **the other** “**Switch()**” function are the **fields** (i.e., “columns”) that I wanted to convert into rows, **in the same order** as in the “**Switch()**” from point 4: “**Q1**”, “**Q2**”, “**Q3**” and “**Q4**”.

Notice that converting columns into rows is the **reciprocal** operation to the one in the **previous** section K.6.4, that addresses **converting rows into columns**.

### **K.6.6 How do I use a Transform operation to show a cross table?**

A **Transform** (click F.10) allows to show your data as a cross table.

Let me show you an example<sup>134</sup> over the Table “T\_Capital\_Rainfall\_Q” (see at the end of

---

<sup>134</sup> This is the Query “K\_Rows\_into\_columns\_2” from file “Company\_Database.accdb”.

this section):

```

TRANSFORM First(Quart_Rainfall) AS GenVals
SELECT Capital AS Cap_City, Cal_Year AS C_Year
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year
PIVOT Quart ;

```

this SQL operation above produces the following output record-list:

K_Rows_into_columns_2					
Capital	Cal_Year	Q1	Q2	Q3	Q4
Beijing	2018	0	4	7,8	17
Washington	2018	12.13	5.67	2.26	12.7

As you may see, this is a cross table of the data from the Table “T\_Capital\_Rainfall\_Q” (copied below). This cross table contains **one** record for each **group** of four quarterly records, where the record contains the values of the **four** quarters.

The guidelines to write a **cross table** with a **Transform** expression in a general case are the following:

1. You identify **one** “**PIVOT**” **expression** (over **input field names**) such that **each** of its **distinct values** will produce **one** “**PIVOT**” **column** (i.e., one “**PIVOT**” **field**). It is quite frequent to use **one input field name** as “**PIVOT**” **expression**.
2. You identify **several** “**GROUP BY**” **expressions** whose **distinct joint values** determine the **number** of **output** records, and also **uniquely identify each output** record. Each “**GROUP BY**” **expression** is built over the **input field names** (without using SQL aggregate functions). It is quite frequent to use as “**GROUP BY**” **expressions** some of the **input fields** that you want to **stay the same** in the **output** records.
3. You identify **one** “**TRANSFORM**” **expression** built over the “**GROUP BY**” expressions, the “**PIVOT**” **expression** and SQL aggregate functions over expressions over the **input field names**. The “**TRANSFORM**” **expression** produces the “**PIVOT**” **field values** (i.e., the **values** in the “**PIVOT**” **columns**). It is quite frequent to use the “**First()**” SQL aggregate function over **one input field name** as “**TRANSFORM**” **expression**.
4. You identify the “**SELECT**” **expressions** that you want. It is quite frequent to use the “**GROUP BY**” **expressions** as the “**SELECT**” **expressions**, because their **joint values** uniquely identify **each output record**.

Let us check how I followed the guidelines in the example above:

1. I identified (as a frequent case) one **input field name**, “**Quart**”, as the “**PIVOT**” **expression**. **Each** of its **distinct values** will produce **one** “**PIVOT**” **column** (i.e., one “**PIVOT**” **field**). Therefore, we will have **four** “**PIVOT**” **fields**, corresponding to the **four distinct values** “**Q1**”, “**Q2**”, “**Q3**”, and “**Q4**”.

2. I identified (as a frequent case) **one input field name** as **each** of the two “**GROUP BY**” **expressions**: “**Capital**” and “**Cal\_Year**”. The **two distinct joint values** of these fields determine the **number** of **output** records, and also **uniquely identify each output** record.
3. I identified (as a frequent case) the “**First()**” SQL aggregate function over the **input field name** “**Quart\_Rainfall**” as “**TRANSFORM**” **expression**.
4. As a frequent case, I identified the “**GROUP BY**” **expressions** “**Capital**” and “**Cal\_Year**” as “**SELECT**” **expressions**.

You can make the most out of the **Transform** operation by converting different data types to *String* through a suitable “**TRANSFORM**” **expression**. This will allow you to present your Query results in an extremely flexible way, combining numbers, strings, dates and *Boolean* values in a seamless way. If you want to know more, you may click “[K.6.13 How do I use String fields to display different data types in the same Query column?](#)”.

The Table “T\_Capital\_Rainfall\_Q” used in the example above is the one of quarterly rainfall measurements in capital cities used in several other examples of this Guide:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

### K.6.7 How do I produce the non-matching records of a Join operation?

A **Select-no\_agg** over a “**LEFT JOIN**” or a “**RIGHT JOIN**” and a “**WHERE**” clause using the “**IsNull()**” function produces the **left** or **right** (respectively) records that do not match the “**ON**” *Boolean* expression.

Let me show it with the following example over the Tables “**T\_House\_Owners**” and

“T\_Car\_Owners” (copied at the end of this section). See the following Query code<sup>135</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
(
  SELECT ID, Name, Address, Not_Null
  FROM
    T_House_owners
  ,
  (
    SELECT Num AS Not_Null FROM T_Numbers WHERE Num=1
  ) AS Nums
) AS Houses
RIGHT JOIN
  T_Car_owners AS Cars
ON Houses.ID = Cars.ID
WHERE IsNull(Not_Null) ;
```

This Query produces the following **output** record-list:

K_Non_matching_Records_1					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
			18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

If you check the content of the Tables below, you will see that these records are exactly the **non-matching** records from the **right** record-list of the “ON” expression.

The Table “T\_Numbers” is an **auxiliary Table** of **integer numbers** (click K.2.2) with just one field named “Num”.

The guidelines to produce the **non-matching records of a Join** in a general case are the following:

1. You do a **Cross-Join** between one **input** record-list and a record-list having only **one** record with **one non-Null** field (it can be exactly the same **Select** over “T\_Numbers” in the example above, highlighted in **green**). The **Cross-Join** is enclosed in a **Select** that exposes **all** the fields from the **Cross-Join** operation, including the field “Num” (changing name to “Not\_Null”, if you want) from “T\_Numbers”.
2. You do a “**RIGHT JOIN**” having as **left** record-list the **Select** from point 1 and as **right input** record-list the one **not used** in point 1. The “ON” expression is the one you want for the **Join**.
3. The “**RIGHT JOIN**” from point 2 is enclosed in a **Select** that exposes all the fields from the **left** and **right input** record-lists. This **Select** has the “**WHERE**” expression “**IsNull(Not\_Null)**”, where “**Not\_Null**” is the field from point 1.

Let us check how I followed these guidelines in the example above:

1. I did a **Cross-Join** between one **input** record-list “T\_House\_owners” and a record-list having only **one** record with **one non-Null** field (it is the **Select** over “T\_Numbers” in the example above, highlighted in **green**). The **Cross-Join** is

<sup>135</sup> This is the Query “K\_Non\_matching\_Records\_1” from file “Company\_Database.accdb”.

- enclosed in a **Select** that exposes **all** the fields from the **Cross-Join** operation, including the field “**Num**” from “**T\_Numbers**” that I renamed to field “**Not\_Null**”.
2. You do a “**RIGHT JOIN**” having as **left** record-list the **Select** from point 1 and as **right input** record-list the one **not used** in point 1 which is “**T\_Car\_owners**”. The “**ON**” expression is the one you want for the **Join**, which in the example is “**Houses.ID = Cars.ID**”.
  3. The “**RIGHT JOIN**” from point 2 is enclosed in a **Select** that exposes all the fields from the **left** and **right input** record-lists. This **Select** has the “**WHERE**” expression “**IsNull (Not\_Null)**”, where “**Not\_Null**” is the name I gave to the “**Num**” field from point 1.

You may be puzzled by my having added a **Cross-Join** operation with just one record from the Table “**T\_Numbers**” (highlighted in **green** above). This **Cross-Join** operation is **mandatory** when you are not sure if the **left** fields may contain **Null** or not. Without it, the Query would produce **matching** records that have **Null** in the field to which you apply the “**IsNull()**” function. Even if you wrote a “**WHERE**” expression checking that **all** the **left** fields are **Null**, it could be the (quite strange) case that the **left** input record-list contained records with all fields having **Null**, and this would again produce a wrong result.

However, if you were **absolutely sure** that one of the **left** fields **cannot** be **Null**, then you can write a simpler Query. You can be absolutely sure of that, for example, when the field is part of the **Primary Key** of the Table. In this example, the field “**ID**” is the **Primary Key** of Table “**T\_House\_Owners**”. Therefore, we could write the following simpler Query<sup>136</sup> that would do the job:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_owners AS Houses
RIGHT JOIN
    T_Car_owners AS Cars
ON Houses.ID = Cars.ID
WHERE IsNull(Houses.ID) ;
```

If you want to know more about the **Join** operators, and what are matching records, you may click “[F.8.2 What are the Join operators?](#)”.

I am copying here the Tables “**T\_House\_Owners**” and “**T\_Car\_Owners**” for your

---

<sup>136</sup> This is the Query “**K\_Non\_matching\_Records\_2**” from file “**Company\_Database.accdb**”.

convenience:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

### K.6.8 How do I write a Full-Outer-Join?

**Full-Outer-Join** is considered undesirable because it produces **Nulls**, which need to be handled properly, weakens indexing and results in worse performance. Avoid **Full-Outer-Join** as much as possible.

In **many** cases what you want is to “**merge**” two record-lists into a single one, and you can get this using “**UNION ALL**” and “**GROUP BY**” instead of using a **Full-Outer-Join**. If you want to “**merge**” two record-lists, you may click “*K.6.9 How do I “merge” two record-lists?*”.

However, on a few occasions you may actually need a **Full-Outer-Join**. You get a **Full-Outer-Join** by doing a “**LEFT JOIN**” plus a “**UNION ALL**” with the **non-matching** records from the **right** record-list. Of course, you can also do it with a “**RIGHT JOIN**” plus a “**UNION ALL**” with the **non-matching** records from the **left** record-list.

Let us then combine the “**LEFT JOIN**” example from *F.8.7.2*, with the **non-matching** records example from *K.6.7*. The result would be the Query<sup>137</sup>:

```
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM F_Join_Left_1
UNION ALL
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM K_Non_matching_Records_1
```

If you ran this Query, you would get the following **output** record-list:

K_Full_Outer_Join_1					
Houses.ID	Houses.Name	Address	Cars.ID	Cars.Name	Car
3	Peter Sellers	345 Bolton St.	3	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	6	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.			
			18	John Welsh	Opel Zafira
			23	Juan Perez	Ford Mustang

If you check the two Tables copied at the end of this section, you will see that this is precisely the result of a **Full-Outer-Join**: the **first two** records are the **matching** records, the **third one** is the **non-matching** record from the **left** record-list and the **fourth** and

<sup>137</sup> This is the Query “K\_Full\_Outer\_Join\_1” from file “Company\_Database.accdb”.



**fifth** are the non-matching records from the **right** record-list.

If you want to check the specific SQL code that produces the **Full-Outer-Join**, I am expanding below the code of the Queries “**F\_Join\_Left\_1**” and “**K\_Non\_matching\_Records\_1**” for your convenience<sup>138</sup>:

```

SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    T_House_Owners AS Houses
LEFT JOIN
    T_Car_Owners AS Cars
ON Houses.ID = Cars.ID
UNION ALL
SELECT Houses.ID, Houses.Name, Address, Cars.ID, Cars.Name, Car
FROM
    (
        SELECT ID, Name, Address, Not_Null
        FROM
            T_House_owners
        ,
        (
            SELECT Num AS Not_Null FROM T_Numbers WHERE Num=1
        ) AS Nums
    ) AS Houses
RIGHT JOIN
    T_Car_owners AS Cars
ON Houses.ID = Cars.ID
WHERE IsNull(Not_Null) ;

```

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

The guidelines to do a **Full-Outer-Join** in a general case are the following:

1. You do a “**LEFT JOIN**” (enclosed in a **Select** operation) between the two **input** record-lists. If one of the two **input** record-lists has one (or more) fields that for sure cannot be Null, that will be the **left input** record-list. The “**ON**” **Boolean** expression is the one that you want for the **Full-Outer-Join**.
2. If the **left** record-list **has** one (or more) fields that for sure cannot be Null, you skip this step. Otherwise, you do a **Cross-Join** between the **left** record-list and a record-list having only **one** record with **one non-Null** field (it can be exactly the same **Select** over “**T\_Numbers**” in the example above, highlighted in **green**). The **Cross-Join** is enclosed in a **Select** that exposes **all** the fields from the **Cross-Join** operation, including the field “**Num**” from “**T\_Numbers**” (with another field name, if you want).
3. You do a “**RIGHT JOIN**” having the same **right input** record-list from point 1. Its **left** record-list is **either** the **Select** from point 2, or the **left input** record-list from point 1, as corresponds (see point 2). The “**ON**” expression **must** be exactly the same as the one of the “**LEFT JOIN**” from point 1.
4. The “**RIGHT JOIN**” from point 3 is enclosed in a **Select** that exposes all the fields from the **left** and **right input** record-lists from point 1. This **Select** has the “**WHERE**” expression “**IsNull(field)**”, where “**field**” is either the “**Num**”

<sup>138</sup> This is the Query “**K\_Full\_Outer\_Join\_2**” from file “**Company\_Database.accdb**”.



field from point 2, or the field from **left input** record-list that for sure **cannot** be **Null**, as corresponds (see point 2).

5. You do a “**UNION ALL**” with the **Select** enclosing the “**RIGHT JOIN**” from point one and the **Select** enclosing the “**RIGHT JOIN**” from point 4.

Let us check how I followed these guidelines in the example above:

1. I did a “**LEFT JOIN**” (enclosed in a **Select** operation) between “**T\_House\_Owners**” and “**T\_Car\_Owners**”. The “**ON**” *Boolean* expression is “**Houses.ID = Cars.ID**”.
2. I knew that both record-lists **have** one field (“**ID**”) that for sure **cannot** be **Null**, but I preferred to show in the example the general case, so I applied this point 2, even when it was not need. I therefore did a **Cross-Join** between the **left** record-list and a record-list having only one record with one non-Null field. I enclosed the **Cross-Join** a **Select** that exposes **all** the fields from the **Cross-Join** operation, including the field “**Num**” from “**T\_Numbers**”, that I renamed to “**Not\_Null**”.
3. I did a “**RIGHT JOIN**” having the same **right input** record-list from point 1 “**T\_Car\_Owners**”. Its **left** record-list is the **Select** from point 2. The “**ON**” expression is **exactly the same** as the one in the “**LEFT JOIN**”.
4. I enclosed the “**RIGHT JOIN**” from point 3 in a **Select** that exposes all the fields from the **left** and **right input** record-lists from point 1. This **Select** has the “**WHERE**” expression “**IsNull(Not\_Null)**”, where “**Not\_Null**” is the “**Num**” field from point 2 that I renamed to “**Not\_Null**”.
5. I did “**UNION ALL**” with the **Select** enclosing the “**RIGHT JOIN**” from point 1 and the **Select** enclosing the “**RIGHT JOIN**” from point 4.

If you want to know more about a **Full-Outer-Join**, you may click “[F.8.8 What is the output record-list of a Full-Outer-Join?](#)”.

If you want to compare a **Full-Outer-Join** with the other **Join** operators, you may click “[F.8.2 What are the Join operators?](#)”.

I am copying here the Tables “**T\_House\_Owners**” and “**T\_Car\_Owners**” for your convenience:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

### K.6.9 How do I “merge” two record-lists?

When I say “**merge**” two record-lists I mean joining records from two input record-lists when both records have **the same values** in some **common specific fields** that **uniquely** identify each record. This is equivalent to say that the **common specific fields** are a

**composite** (candidate) **Key** of **each** of the input record-lists.

Let me show this to you by merging the same Tables “**T\_House\_Owners**” and “**T\_Car\_owners**” (copied at the end of this section) that I used for the **Full-Outer-Join**. One way of merging both Tables is doing a **Full-Outer-Join** (click *K.6.8*), and then merging the two “**ID**” fields into a single field. This would be done with the following Query<sup>139</sup>:

```
SELECT Iif(IsNull(Houses.ID), Cars.ID, Houses.ID) AS ID
      , Houses.Name AS Houses_Name, Houses.Address AS Houses_Address
      , Cars.Name AS Cars_Name, Cars.Car AS Cars_Car
FROM K_Full_Outer_Join_1
```

The Query “**K\_Full\_Outer\_Join\_1**” from *K.6.8* performs the **Full-Outer-Join**. The merging of the two “**ID**” fields is done with the “**Iif()**” and “**IsNull()**” functions. The other fields from the **Full-Outer-Join** are just renamed. The result of the Query above is:

K_Merge_Tables_1				
ID	Houses_Name	Houses_Address	Cars_Name	Cars_Car
3	Peter Sellers	345 Bolton St.	Peter Sellers	Renault Clio
6	Xi Liu	1342 Main St.	Xi Liu	Ford Focus
14	Xi Liu	580 Beacon St.		
18			John Welsh	Opel Zafira
23			Juan Perez	Ford Mustang

You may see that we achieved the desired result.

However, the **Full-Outer-Join** operation has some disadvantages (click *F.8.2*), and you should avoid it as much as possible. This is why I am showing you here how to merge two record-lists using “**UNION ALL**” and “**GROUP BY**”.

The following Query<sup>140</sup> achieves **exactly the same** result just using “**UNION ALL**” and “**GROUP BY**”:

```
SELECT ID, Max(Name) AS Houses_Name, Max(Address) AS Houses_Address
      , Max(C_Name) AS Cars_Name, Max(Car) AS Cars_Car
FROM
(
  SELECT ID, Name, Address, Null AS C_Name, Null AS Car
  FROM T_House_Owners
  UNION ALL
  SELECT ID, Null, Null, Name, Car
  FROM T_Car_Owners
)
GROUP BY ID
```

The guidelines to **merge two record-lists** in a general case are the following:

1. You enclose **each** of the two record-lists that you want to merge in a **Select** operation. The **field(s) that identify** each record in both record-lists that you want to merge (that must be completely equivalent one-to-one) are **placed in the same**

<sup>139</sup> This is the Query “**K\_Merge\_Tables\_1**” from file “**Company\_Database.accdb**”.

<sup>140</sup> This is the Query “**K\_Merge\_Tables\_2**” from file “**Company\_Database.accdb**”.

**output field positions.**

2. The **remaining** fields are placed on **output field** positions **mutually disjoint** between the two record-lists.
3. The **values** of the fields in positions that correspond to the **other** record-list are set to **Null**.
4. Having done the above, you do a “**UNION ALL**” over both **Select** operations.
5. Having done the above, you enclose the **Union** operation in a **Select-group\_by\_aggreg** operation, where the “**GROUP BY**” **expressions** are the fields from point 1 above. These fields from point 1 are placed **as such** as “**SELECT**” **expressions** of the **Select-group\_by\_aggreg**. Each of the other “**SELECT**” **expressions** is the “**Max ()**” SQL aggregate function over the corresponding fields.

Let us check how I followed these guidelines in the example above:

1. I enclosed **each** of the two record-lists (the Tables “**T\_House\_Owners**” and “**T\_Car\_Owners**” in a **Select** operation. The “**ID**” **field that identifies** each record in both record-lists (that is completely equivalent one-to-one) are **placed in the same output field position**: they are both placed in position **one** (left to right).
2. The **remaining** fields are placed on **output field** positions **mutually disjoint** between the two record-lists. I placed the fields “**Houses.name**” and “**Houses.Address**” in positions **two** and **three**, and the fields “**Cars.Name**” and “**Cars.Car**” in positions **four** and **five** (left to right), respectively.
3. The **values** of the fields in positions that correspond to the **other** record-list are set to **Null**. I did this easily setting to “**Null**” the corresponding “**SELECT**” **expressions** of each of the two innermost **Select** operations.
4. Having done the above, I did a “**UNION ALL**” over both **Select** operations.
5. Having done the above, I enclosed the **Union** operation in a **Select-group\_by\_aggreg** operation, where the “**GROUP BY**” **expressions** is the field from point 1 above: this is the field “**ID**”. This field “**ID**” from point 1 I placed **as such** in the first “**SELECT**” **expression**. Each of the other “**SELECT**” **expressions** is the “**Max ()**” SQL aggregate function over the corresponding fields.

Notice that you **must** use the SQL aggregate function “**Max ()**” (or “**Min ()**”) and you **must not** use “**First ()**” (nor “**Last ()**”). The reason is that “**First ()**” (and “**Last ()**”) return the value from an **unknown** record **even if it is Null**. However, “**Max ()**” (and “**Min ()**”) always return a **non-Null** value (as long as there is one) which is precisely what we want.

I am copying here the Tables “**T\_House\_Owners**” and “**T\_Car\_Owners**” for your

convenience:

T_House_owners		
ID	Name	Address
3	Peter Sellers	345 Bolton St.
6	Xi Liu	1342 Main St.
14	Xi Liu	580 Beacon St.

T_Car_owners		
ID	Name	Car
3	Peter Sellers	Renault Clio
6	Xi Liu	Ford Focus
18	John Welsh	Opel Zafira
23	Juan Perez	Ford Mustang

### K.6.10 How do I produce a weighted average?

Doing an “INNER JOIN” with a **Select-group\_by\_aggreg** and the “**Sum()**” SQL aggregate function.

Let me show you an example<sup>141</sup> over the Table “T\_Capital\_Rainfall\_Q” (see at the end of this section):

```
SELECT  Q.Capital, Q.Cal_Year, Q.Quart
        , Quart_Rainfall/Yearly_Rainfall AS Yearly_fraction
FROM    T_Capital_Rainfall_Q AS Q
INNER JOIN
(
  SELECT Capital, Cal_Year, Sum(Quart_Rainfall) AS Yearly_Rainfall
  FROM T_Capital_Rainfall_Q
  GROUP BY Capital, Cal_Year
) AS Y
ON (Q.Capital=Y.Capital) AND (Q.Cal_Year=Y.Cal_Year)
```

The output record-list would be:

K_Weighted_Average			
Capital	Cal_Year	Quart	Yearly_fraction
Beijing	2018	Q1	0.0%
Beijing	2018	Q2	13.9%
Beijing	2018	Q3	27.1%
Beijing	2018	Q4	59.0%
Washington	2018	Q1	37.0%
Washington	2018	Q2	17.3%
Washington	2018	Q3	6.9%
Washington	2018	Q4	38.8%

As you may see, the field “**Yearly\_fraction**” has the **weights** of the quarterly rainfall within each calendar year. If you wanted to do a weighted average, you only need to multiply the **value** that you want to **weight** by “**Yearly\_fraction**”.

<sup>141</sup> This is the Query “K\_Weighted\_Average” from file “Company\_Database.accdb”.

The guidelines to do a **weighted average** in a general case are the following:

1. You enclose the **input** record-list in a **Select-group\_by\_aggreg** operation. The “**GROUP BY**” **expressions** assign each record to a **group** within which you will do a weighted average. **Each** distinct set of values of the “**GROUP BY**” **expressions** identifies **one** such **group**. **Each group** has **one total** value that will be the **divisor(s)** (the number in the **lower** part of a division) of the **weights** in each **group**. The total value is obtained with the “**Sum ()**” SQL aggregate function over the value of each dividend (the number in the **upper** part of a division) of each weight.
2. You do an **Inner-Join** of the **Select-group\_by\_aggreg** (from point 1) with the **input** record-list, where the “**ON**” **expression** is a set of **pairwise** equality expressions (combined with “**AND**”) between the **values** of the “**GROUP BY**” **expressions** coming from the **Select-group\_by\_aggreg** and the **input** record-list.
3. The “**SELECT**” **expressions** of the **Select** enclosing the **Inner-Join** operation (from point 2) are the “**GROUP BY**” **expressions** (that identify each **group** of weighted averages), the **expression** that identifies each individual value within each **group** and the **weight**, which is the division between each **dividend** (this is, the **argument** of the “**Sum ()**”) and its **divisor** (this is, the **result** of the “**Sum ()**”).

Let us check how I followed these guidelines in the example above:

1. I enclosed the **input** record-list “**T\_Capital\_Rainfall\_Q**” in a **Select-group\_by\_aggreg** operation. The “**GROUP BY**” **expressions** have been “**Capital**” and “**Cal\_Year**”. The distinct values of these two **expressions** identify each **group** within which I will do a weighted average. **Each group** has **one total** value that will be the **divisor(s)** (the number in the **lower** part of a division) of the **weights** in each **group**. The total value is obtained with the “**Sum ()**” SQL aggregate function over “**Quart\_Rainfall**”, which is the value of each dividend (the number in the **upper** part of a division) of each weight.
2. I did an **Inner-Join** of the **Select-group\_by\_aggreg** (from point 1) with the **input** record-list “**T\_Capital\_Rainfall\_Q**”, where the “**ON**” **expression** is a set of **pairwise** equality expressions (combined with “**AND**”) between the **values** of the “**GROUP BY**” **expressions** coming from the **Select-group\_by\_aggreg** (named “**Y**” in this example) and the **input** record-list (named “**Q**”):
 

```
ON (Q.Capital=Y.Capital) AND (Q.Cal_Year=Y.Cal_Year)
```
3. I chose the following “**SELECT**” **expressions** of the **Select** enclosing the **Inner-Join** operation (from point 2):
  - The “**GROUP BY**” **expressions** (that identify each **group** of weighted averages): “**Capital**” and “**Cal\_Year**”.
  - The **expression** that identifies each individual value within each **group**: “**Quart**”
  - The **weight**, which is the division between each **dividend** (this is, the **argument** of the “**Sum ()**”) and its **divisor** (this is, the **result** of the “**Sum ()**”):

`“Quart_Rainfall/Yearly_Rainfall”`

Notice that in case the **group** of values over which “**Sum()**” is applied contains one or more **Null**, all of the **Null** are **ignored** in the computation of the weights, because the “**Sum()**” SQL aggregate function ignores **Nulls**. If you wanted to take the **Null** into account (e.g., considering them as the value zero), it is easy to modify the expression that is an argument to “**Sum()**”. For example, you could do:

```
Sum(Iif(IsNull(Quart_Rainfall), 0, Quart_Rainfall)) AS Yearly_Rainfall
```

If you want to know more about “**INNER JOIN**” and/or “**Sum()**”, you may click “[F.8.6.2 What are the output records of an “INNER JOIN”?](#)” and “[F.7.18.5 What are the “Sum\(\)” and “Avg\(\)” SQL aggregate functions?](#)”.

The Table “T\_Capital\_Rainfall\_Q” used in the example above is the one of quarterly rainfall measurements in capital cities used in several other examples of this Guide:

T_Capital_Rainfall_Q			
Capital	Cal_Year	Quart	Quart_Rainfall
Beijing	2018	Q1	0
Beijing	2018	Q2	4
Beijing	2018	Q3	7.8
Beijing	2018	Q4	17
Washington	2018	Q1	12.13
Washington	2018	Q2	5.67
Washington	2018	Q3	2.26
Washington	2018	Q4	12.7

### K.6.11 How do I design a data check Query?

A data check Query is composed of a long list of **Union** operations. Each **Select** operation in the long list of **Union** operations checks for errors on a specific Table, or on a specific set of Tables. Each **Select** operation takes a number of records from a Table (or a number of joined records from several Tables) and performs a number of error checks over each of them. For **each** erroneous record or joined-records the Query will produce **one** output record with an informative error message.

It is important to highlight that the Query should avoid retrieving the same records several times, to prevent the data check Query from becoming very slow (click [K.7](#)). Therefore, for each record, or joined records, that you retrieve, do all the checks over it/them in a given **Select** operation and **do not** have several **Select** operations, each checking a different error over **the same record-list**.

Since there are many different error checks that you may need to do, over records with very different record-types, the best is that the data check Query has only **one** output field of **Short Text** type. In this way, the output of each and every **Select** operation are also records with just one **Short Text** field, and the **Union** operations binding all these **Select** operations work perfectly.

To avoid the data check Query becoming very slow, **do not** check **all** data in your Tables.



In order to limit the amount of data checked, you should design the data check Query to check only the “**most recent**” Table data (e.g., the latest “**n**” invoices, or all the invoices in the “**n**” last years). The concrete interpretation of “most recent” can be hardwired in the Query, where the value of “**n**” is fixed in the Query code, or you can add a parameter to the Query indicating the extent of Table data that should be checked. You could for example add a **Date/Time** parameter that indicates to check all data more recent that the date provided as the parameter. The approach I personally like best is to have the Query without a parameter and design it to check the data from the last “**n**” years. The value of “**n**” you choose depends on the yearly volume of data added to your database, but frequent values are between 1 and 5 years.

Some examples of error checks that you may find useful are:

- In a Table of contracts, check that no person has two contracts that overlap in time. This can be achieved doing a self-join (i.e., a join of the Table records with themselves), where the “**ON**” *Boolean* expression requires that the person name in the two input record-lists is the same, and the contract dates overlap. You then have to remove the matching of a contract with itself, which is not difficult, and you can thus detect overlapping contracts. The output-expression has to be designed to produce one single **Short Text** field, with one error message out of the different fields of each erroneous contract record.
- In a Table of project expenses, where each expense may be **partially** charged to a project, check that the expense amounts charged to different projects add up to the total amount of the expense. You can do this by first selecting **only** the records that correspond to expenses partially charged, for example, by putting a “**WHERE**” clause requiring that the “Total\_amount” of the expense and the “Amount\_charged” to the project are different. You then add up the values of “Amount\_charged” for each expense using a “**GROUP BY**” clause over the field that identifies the expense (e.g., “Invoice\_number”, or whatever field identifies each expense). Then, you add a “**HAVING**” clause requiring that the **sum** of partials is different from the total), and this gives you the expenses that do not add up correctly. The “**HAVING**” clause would be:

```
HAVING Sum(Amount_charged) <> First(Total_amount)
```

The “**SELECT**” **expression** has to be designed to produce one single **Short Text** field, with one error message out of the different fields of each erroneous contract record.

- Between a Table of projects and a Table of project expenses, check that the date of every project expense is within the start and end dates of the project. This can be achieved doing a “**LEFT JOIN**” between the project expenses Table and the projects Table where the “**ON**” *Boolean* expression requires that the project in each record is the same. Then you add a “**WHERE**” clause requiring that the expense date is outside of the project start and end dates. The “**SELECT**” **expression** has to be designed to produce one single **Short Text** field, with one error message out of the different fields of each erroneous contract record.

You must **run** the data check Query quite frequently (e.g., once a day), to detect as soon as possible any possible data error in your database. Also, you should run the data check Query when you have detected an error in the results of a test and proven Query, because

very likely the error in test and proven Queries comes from erroneous Table data.

### **K.6.12 How do I get the exact record ordering I want?**

A **Select** with “**DISTINCT**” and a **Transform** have severe limitations on how you can design its “**ORDER BY**” expressions (click *F.7.12* and *F.10.8*). For the case of the **Select**, this is trivial to solve by enclosing it inside another **Select** operation, where you can design the “**ORDER BY**” expressions as you want. For the case of a **Transform**, it cannot be solved: each “**ORDER BY**” expression can only be exactly the same as one of the “**GROUP BY**” expressions. However, you can create specific “**GROUP BY**” expressions in the **Transform** to order your records (click *K.6.12.4*).

For the **Select** operation, there are many useful ways of writing the list of “**ORDER BY**” expressions. I show a few of them in:

- “*K.6.12.1 How do I order over several expressions?*”
- “*K.6.12.2 How do I order over only one expression?*”
- “*K.6.12.3 How do I order over user-defined functions?*”
- “*K.6.12.4 How do I order over “GROUP BY” expressions?*”
- “*K.6.12.5 How do the four ordering approaches compare among themselves?*”

#### **K.6.12.1 How do I order over several expressions?**

Designing **several** expressions that placed in your “**ORDER BY**” clause will produce the record-ordering that you want.

Imagine that you have the Table “T\_Contracts” with fields “Pers” (person name), “Branch” (the branch where the person works), “Salary” and “RenD” (contract renewal date). Imagine that you want the following order: people with a salary higher than \$100,000, followed by people with a salary lower than \$40,000, followed by the other people. Order the people in the first group by descending salary. Order the people in the second group by ascending salary. Order the people in the third group by ascending contract renewal date. An “**ORDER BY**” expression<sup>142</sup> that does this is:

```
ORDER BY Switch( Salary > 100000, 1
                , Salary < 40000, 2
                , True           , 3)
        , Switch( Salary > 100000, -Salary
                , Salary < 40000, Salary
                , True           , RenD )
```

Notice how ordering over one (or more) initial “**Switch()**” function(s) allows to order records **in arbitrary groups**. Notice how ordering over a last “**Switch()**” function allows to use a different ordering criterion **within** each group. Notice how the records that have the same values on the first “**n**” ordering criteria of the “**ORDER BY**” clause are ordered among themselves according to criterion “**n+1**”.

#### **K.6.12.2 How do I order over only one expression?**

Designing **only one** expression that placed in your “**ORDER BY**” clause produces the

<sup>142</sup> This is the Query “K\_Order\_by\_1” from file “Company\_Database.accdb”.



record-ordering that you want.

Using the same example from the previous subsection *K.6.12.1*, you could write the following “**ORDER BY**” expression<sup>143</sup> that produces the same ordering:

```
ORDER BY Switch( Salary > 100000, 100+Salary/100000
               , Salary < 40000 , 50-Salary/100000
               , True           , -RenD/100000) DESC
```

### **K.6.12.3 How do I order over user-defined functions?**

Writing user-defined VBA functions that takes as arguments the fields that you need for your ordering and produces a value according to the ordering that you want.

### **K.6.12.4 How do I order over “GROUP BY” expressions?**

In a **Transform** operation, each “**ORDER BY**” expression can only be **exactly the same** as **one** of the “**GROUP BY**” expression(s). This is a very severe limitation, but you can create **specific** “**GROUP BY**” expression(s) to order your records. You have to do it so it will **not** modify the **grouping** of the records that you want to produce. This is trivial to do if the ordering “**GROUP BY**” expression(s) are built using **the other** “**GROUP BY**” expression(s) as their elements.

Doing this you get a **flexible** way of ordering the records of a **Transform** operation. My advice is that you **include each and every** of the ordering “**GROUP BY**” expression(s) as a “**SELECT**” expression of the **Transform** operation. In this way, the user can **see** (as a field in the Query results) the values over which the records are ordered, and also, she/he can modify the record sorting interactively (click *H.2.3*).

### **K.6.12.5 How do the four ordering approaches compare among themselves?**

The four ordering approaches shown in *K.6.12.1*, *K.6.12.2*, *K.6.12.3* and *K.6.12.4* are complementary and not disjoint. You can therefore use the one most convenient for **each** specific SQL operation, as well as **combinations** of them, as you see fit.

Regarding simplicity of coding and code maintenance, the simplest approach is ordering over **several expressions**, followed by ordering over **one expression**, ordering over **user-defined functions** and finally ordering over “**GROUP BY**” expression(s) which is the most complex.

The advantages of ordering over several expressions or over a user-defined function is that you can combine ordering over *String* fields combined with **numeric-like** fields. However, trying to order over a combination of *String* fields and **numeric-like** fields over only one expression has very **severe limitations**.

The advantage of using only one expression or user-defined function, is that you can use the ordering value as an **output field** called for example “Ord”. This allows you to change the ordering (e.g., based on other fields) while you are seeing the Query results in “**Datasheet View**” (click *H.2.3*) and recover the original ordering at any moment by ordering on the field “Ord”. It is true that you can always recover the original ordering by running the Query again, without needing an “Ord” field, but some Queries are slow, and

---

<sup>143</sup> This is the Query “**K\_Order\_by\_2**” from file “**Company\_Database.accdb**”.

doing this would have a performance penalty.

The advantage of ordering over a user-defined function is that it is the most flexible and you can change the ordering by modifying your VBA code, without having to modify your SQL code.

The advantage of ordering over “**GROUP BY**” **expression(s)** is that you can flexibly order the output records of a **Transform** operation.

### **K.6.13 How do I use *String* fields to display different data types in the same Query column?**

This section indicates how to **display values** from **different data types** in the **same field** (in **different records**) in **Query** results (i.e., a “column” displaying values of different data types). If what you want is to **store**, in **one Table field** (“column”), a **list of values**, you may click “*K.2.8 How do I store a list of values, instead of a single value, in a Table field?*”.

You display values from different data types in the same field (in **different records**) by **converting** all the values to the **String** data type. In order to display the values properly, you use the “**Format()**” function and Program Flow functions (click *M.15*) to convert all numeric-like<sup>144</sup> values to **String** data type, while formatting each specific value (**Boolean**, **Integer**, **Date**) in the required way.

For example, imagine that you want the following **output** record-list<sup>145</sup>:

<b>K_Output_formatting</b>			
<b>Data_element</b>	<b>Zone_A</b>	<b>Zone_B</b>	<b>Zone_C</b>
State	California	Spain	Germany
Top Employee	Cindy Carpenter	Juan del Rio	Steffan Schmidt
Employee Sales	230,945.78	204,546.34	203,478.52
Company Sales	25,254,478.85	27,234,497.42	30,234,478.78
Top Sales Day	3-Jan-2025	23-Apr-2025	17-Nov-2025
Labor Conflict	Yes	No	No

as you may see, each column **appears** to combine values from **String**, **Double**, **Date** and **Boolean** data types. In fact, all these values are of **String** data type.

Converting all values to **String** is extremely useful and flexible to present the **final** results of your Queries. However, if you do this, you should be aware of the following problems:

- **No further processing**

If you want to use the results of this Query in another Query, you can do it, but since you have values from **different** data types in the **same** field, the processing you can do is extremely limited. For this reason, this technique of combining values of different data types converting them to **String** is mainly used to present the **final** results of a Query, that will not have further processing.

<sup>144</sup> Remind that numeric-like data types include **Date** and **Boolean**.

<sup>145</sup> This is the Query “**K\_Output\_formatting**” from file “**Company\_Database.accdb**”.

- **Same Alignment**

All the values in **each column** must have the **same alignment**. This is, within the same column, you **cannot** have **left justification** in some values **and right justification** in other values: all the values in each column are **either right** justified, **left** justified, or whatever other justification you configure. The default *String* field justification is **left** justified. If you want a field to be **right** justified just configure the field property “**Format**” as “**Standard**”. If you want to know how to do this, you may click “*H.7 How do I configure the column text alignment in a Table/Query/Form?*”.

- **String ordering**

All the values are *String*. Therefore, if you sort the results over a given column, the records will be ordered based on *String* ordering, and not according to the “apparent” value. For example, if you sort a column with values “**20-Aug-2018**”, “**15-Dec-2019**” and “**3-Jan-2020**” in “AtoZ” ordering you will get:

15-Dec-2019
20-Aug-2018
3-Jan-2020

which is the alphabetical ordering used in *String*, which is clearly different from the sorting if these values were *Date*.

- **Copy and Paste into Excel**

When you copy and paste values into Excel (or any other application that enforces data types), all the values will be passed as **text strings**, even if they look like numbers, dates or *Booleans*, because the default pasting maintains the original data types. Therefore, if you try to sum in Excel a column of values that look like numbers, the result will be **zero**, because they are all text strings.

What you can do is paste as unformatted text, and then Excel will interpret each string of unformatted text and will most likely (but not always) paste the values correctly.

As a conclusion, I would like to point out that databases have lots of advantages over spreadsheets, but one disadvantage is that, in principle, the values in a given Query column in “**Datasheet View**” must **all** have the **same data type** and the **same formatting**. This greatly restricts the possibilities of designing your Query results in a way that show the information in the way you want. However, as we have seen along this section, by converting all values to *String*, you can combine in the same Query column values of very different data types, achieving a very high data-presentation flexibility.

### **K.6.14 What are useful tricks with SQL aggregate functions?**

MS-Access does not allow to code user-defined SQL aggregate functions in VBA. You can therefore only use the built-in SQL aggregate functions (click *F.7.18*). This section explains how to use the available built-in SQL aggregate functions to produce results that you may need, as if **other** SQL aggregate functions **would** exist:

- “**AllNull (exp ())**”:

Returns **True** if **all** the results of the expression in its argument are **Null**.

You may click “*K.6.14.1 How do I use the “AllNull ()” SQL aggregate function?*”.

- “**AllEqual** (exp ())”:  
Returns *True* if **all** the results of the expression in its argument have the same value.  
You may click “[K.6.14.2 How do I use the “AllEqual \(\)” SQL aggregate function?](#)”.
- “**Or** (exp ())” and “**And** (exp ())”:  
Returns the “**OR**” or “**AND**” (respectively) over **all** the results of the expression in its argument.  
You may click “[K.6.14.3 How do I use the “Or \(\)” or “And \(\)” SQL aggregate functions?](#)”.

#### K.6.14.1 How do I use the “**AllNull** ()” SQL aggregate function?

Using the expression “**Count** (\*) <> **Count** (exp (INOUT-field-names))”.

MS-Access **does not** provide the “**AllNull** ()” SQL aggregate function to return *True* when **all** the records in the **group** of records produce **Null** in a given expression “exp (INOUT-field-names)”.

However, if you use the expression “**Count** (\*) <> **Count** (exp (INOUT-field-names))” it produces **the same result**.

Let me show you an example<sup>146</sup> over the Table “T\_Capital\_Rainfall\_Q” (see at the end of K.6.10):

```
SELECT Capital, Cal_Year, Count(*) <> Count(Quart_Rainfall) AS Any_Null
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year
```

The **output** record-list is the following one, with all values of the field “**Any\_Null**” being *False* (remind that *False* is numerically represented as “0”):

K_AllNull		
Capital	Cal_Year	Any_Null
Beijing	2018	0
Washington	2018	0

because in “T\_Capital\_Rainfall\_Q” there is no **Null** in the field “**Quart\_Rainfall**”.

If you want to know more about “**Count** (\*)” and “**Count** ()”, you may click:

- “[F.7.18.1 What is the “Count \(\\*\)” SQL aggregate function?](#)”
- “[F.7.18.2 What is the “Count \(\)” SQL aggregate function?](#)”.

#### K.6.14.2 How do I use the “**AllEqual** ()” SQL aggregate function?

This subsection also answers the question:

- **How do I check if all field values in a group of records are the same?**

<sup>146</sup> This is the Query “K\_AnyNull” from file “Company\_Database.accdb”.

You do it using the expression:

`“Min(exp(INOUT-field-names)) = Max(exp(INOUT-field-names))”`

MS-Access **does not** provide the `“AllEqual()”` SQL aggregate function to return *True* if all the values in a given field value-list are the same in each **group of records**. However, using the expression `“Min(exp()) = Max(exp())”` produces the same result.

Let me show you an example<sup>147</sup> over the Table `“T_Capital_Rainfall_Q”` (see at the end of *K.6.10*):

```
SELECT Capital, Cal_Year
      , Min(Quart_Rainfall) = Max(Quart_Rainfall) AS Equal_values
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year
```

The **output** record-list is the following one, with all values of the field `“Equal_values”` being *False* (remind that *False* is numerically represented as `“0”`):

K_AllEqual_1		
Capital	Cal_Year	Equal_values
Beijing	2018	0
Washington	2018	0

because in Table `“T_Capital_Rainfall_Q”` there is no capital city and year were all the `“Quart_Rainfall”` values are the same.

Notice that in case the list of field values of a **group** of records contains one or more **Null**, the **Null** are **ignored**, because both `“Min()”` and `“Max()”` SQL aggregate functions ignore **Nulls**. If you want to check if all the values are the same, and that there are no **Nulls**, you can combine the expression in this subsection with the one of the previous one. The result would be the expression:

```
(Min(Field_name) = Max(Field_name)) AND (Count(*) = Count(Field_name))
```

In the previous example, the resulting SQL code<sup>148</sup> would be:

```
SELECT Capital, Cal_Year
      , (Min(Quart_Rainfall) = Max(Quart_Rainfall))
      AND (Count(*) = Count(Quart_Rainfall)) AS Equal_values
FROM T_Capital_Rainfall_Q
GROUP BY Capital, Cal_Year
```

If you want to know more about `“Min()”` and `“Max()”`, you may click [“F.7.18.4 What are the “Min\(\)” and “Max\(\)” SQL aggregate functions?”](#).

### K.6.14.3 How do I use the `“Or()”` or `“And()”` SQL aggregate functions?

Using `“Min()”` to produce the same result as `“Or()”` and using `Max()”` to produce the same result as `“And()”`.

MS-Access provides neither the `“Or()”` (computing an **“OR”** over **all** the values of a field

<sup>147</sup> This is the Query `“K_AllEqual_1”` from file `“Company_Database.accdb”`.

<sup>148</sup> This is the Query `“K_AllEqual_2”` from file `“Company_Database.accdb”`.

value-list in each **group of records**) SQL aggregate function nor the “**And()**” (computing and “**AND**” over **all** the values of a field value-list in each **group of records**) SQL aggregate function. However, because **True** is internally represented as “**-1**” and **False** is internally represented as “**0**” (click *D.4.7*), the result of “**Min()**” over **Boolean** values is equivalent to “**Or()**” and the result of “**Max()**” over **Boolean** values is equivalent to “**And()**”.

Notice that in case the field value-list of a **group of records** contains one or more **Null**, the **Null** are **ignored**, because both “**Max()**” and “**Min()**” SQL aggregate functions ignore **Nulls**. If you want to do an “**Or()**” or “**And()**” making sure that there are no **Nulls**, you should make an “**AND**” with the expression:

```
Count(*) = Count(Field_name)
```

If you want to know more about “**Min()**” and “**Max()**”, you may click “*F.7.18.4 What are the “Min() and “Max() SQL aggregate functions?”*”.

## **K.7 Why and how do I design a fast database and fast Queries?**

Because you may have **very big differences** in processing time between doing an efficient design and a non-efficient design. If you want a quite clear example about this, you may click *K.7.2.2*. In respect to how to do it, you will find a number of indications in the following sections within in this chapter.

MS-Access contains a **Query optimizer**, which is a software tool that analyzes the logic underlying each Query’s SQL code and decides what is the most **efficient** (i.e., fastest) way to produce that result. This implies that even if you write not very efficient SQL Queries, the Query optimizer may actually make them run fast. However, it is quite risky to rely on the intelligence of the Query optimizer because if you do a **bad database design** and/or **bad Query designs**, your Queries may run **really slow**.

For **small amounts of data** (a few records and fields) you may not **perceive** the advantages of doing efficient designs because the processing time is very small. For example, if you have a Query processed in 50 milliseconds and another one in ten times more (i.e., 500 milliseconds), you will perceive both Queries as “instantaneous”. However, if one takes 5 seconds and the other 50 seconds to be processed, which is also ten times more, you will perceive a **huge difference** between both Queries.

Along this chapter *K.7* I will explain how do to an **efficient** database design and an **efficient** Query design. The SQL examples will be presented over four Tables called “**T\_Effic\_10**”, “**T\_Effic\_100**”, “**T\_Effic\_1000**” and “**T\_Effic\_3000**”. Each of these Tables has a **Number-Long Integer** field called “**Num**” and a **Short Text** field called “**Strng**”. The first Table has 10 records, the second one has 100 records, the third one has 1,000 records and the fourth one has 3,000 records. The content of these four Tables is outlined as

follows:

T_Effic_10		T_Effic_100		T_Effic_1000		T_Effic_3000	
Num	Strng	Num	Strng	Num	Strng	Num	Strng
0	0000	0	0000	0	0000	0	0000
1	0001	1	0001	1	0001	...	...
...	...	...	...	...	...	999	0999
9	0009	98	0098	99	0099	1,000	1000
		99	0099	100	0100	...	...
				...	...	2,000	2000
				999	0999	2,001	2001
						...	...
						2,999	2999

If you want to know more about this, you may click:

- [“K.7.1 How do I design a fast database?”](#)
- [“K.7.2 How do I design faster Select operations over Queries?”](#)
- [“K.7.3 How do I design faster Select operations over Tables?”](#)
- [“K.7.4 How do I design faster Union operations?”](#)
- [“K.7.5 How do I design faster Join operations?”](#)
- [“K.7.6 How do I design a faster Select-group by when I have bound values in all my records?”](#)

### K.7.1 How do I design a fast database?

The summary of recommendations for making a fast database design is:


- **Add a Primary Key**  
You should add a (simple or composite) **Primary Key** to (almost) all your Tables. If you want to know more about **Primary Keys**, you may click [“C.10.1 What is the Primary Key of a Table?”](#) and [“D.6 How do I configure the Primary Key field\(s\) of a Table?”](#). In the example “T\_Effic\_” Tables (described in K.7) you have two simple candidate **Keys**: “Num” and “Strng” (either can be used).
- **Index (almost) all your Table fields,**  
You should index **every** Table field that **may be** included in a “**WHERE**”, “**ON**” or “**HAVING**” *Boolean* expression, or in an “**ORDER BY**” or “**GROUP BY**” expression. This basically means indexing (with or without duplicate values, as corresponds) **almost all** your Tables’ fields. In the example Tables “T\_Effic\_” (described in K.7) you should index without duplicates the field that was not configured as the **Key** field. If you want to know more about indexing, you may click [“C.8 What is indexing?”](#) and [“D.7 How do I add simple and/or composite index\(es\) to a Table?”](#).
- **Add composite indexes when possible**  
You should add a composite index over **every group** of Table fields that **may be jointly** included in a “**WHERE**”, “**ON**” or “**HAVING**” *Boolean* expression, or in an



“**ORDER BY**” or “**GROUP BY**” expression. This basically means adding a composite index to every group of fields that may have some **joint semantics**. In the example Tables “**T\_Effic\_**” (described in K.7) it does not make much sense to add a composite index because there is no joint semantics between the two fields. If you want to know more about composite indexes, you may click “C.8.3.1 What are simple indexes and composite indexes?” and “D.7.2 How do I add composite (and simple) indexes to a Table?”.

- **Configure as many Relationships as makes sense**

You should configure as many Relationships with referential integrity as it is possible. Relationships not only make your Queries run faster, but they also provide guarantees of integrity for your database data. If you want to know more about Relationships, you may click “C.11 What is a Relationship?” and “D.9 How do I create and configure my Table Relationships?”.

MS-Access has a tool that checks your Tables and makes suggestions in respect to adding indexes and/or Relationships. Click on “**Database Tools**” from the Ribbon-bar, and then click on the **Analyze Performance** “” icon. This will open a dialog box where you can select different database elements (Tables, Queries, ...) by clicking on the corresponding **tabs**. Once you have selected a **tab**, you may individually select objects (ticking their checkbox), or rather, select **all** of them clicking the “**Select All**” button. When you are done selecting objects, click the “**OK**” button. MS-Access will analyze the selected objects and will present a list of suggestions to improve performance.

## **K.7.2 How do I design faster Select operations over Queries?**

You may click:

- “K.7.2.1 Why should I use “DISTINCT”, “UNION” and “ORDER BY” only if needed?”
- “K.7.2.2 Why should I use the most restrictive “WHERE” and “HAVING” expressions in my Select operations?”
- “K.7.2.3 Why should I mainly use comparison and logical operators in my “WHERE” and “HAVING” expressions?”

### **K.7.2.1 Why should I use “DISTINCT”, “UNION” and “ORDER BY” only if needed?**

Because the “**DISTINCT**”, “**UNION**” and “**ORDER BY**” require that MS-Access processes the whole **output** record-list to remove duplicates or to order the records (respectively) which will make your Query slower. If the record-list is long and/or with many fields, the required processing time may be **considerable**, in particular if there is **no indexing** (click C.8).

You should only use “**DISTINCT**” if the **output** record-list **may have** duplicate records and you **do not want** duplicate records. This exact same advice applies to using “**UNION**” in **Union** operations.

You should only use “**ORDER BY**”, **if required**, in the **outermost Select** operation of the Query. **Never** use it in **inner Select** operations within your Query because the ordering



will be lost in the enclosing SQL operations and therefore you are making your Query slower for no reason. An “**ORDER BY**” may be useful in your outermost **Select** operation only if this is Query is invoked by the user (i.e., it is **not only** an auxiliary Query).

### K.7.2.2 Why should I use the most restrictive “WHERE” and “HAVING” expressions in my Select operations?

Because the smaller are the record-list operands of SQL operations, the faster the processing will be. If you keep processing huge record-lists all along your Query code, only to be filtered in the outermost **Select** operation your Queries may become really slow.

Let me show the difference using an example over the Tables “**T\_Effic\_**” (described in K.7). The following Query<sup>149</sup> takes in my laptop **FOURTY FIVE seconds** to run!!

```

SELECT T_A.Num, T_B.Num, T_C.Num
FROM
    T_Effic_1000 AS T_A
, -- "Comma" is the Cross-Join operator
    T_Effic_1000 AS T_B
, -- "Comma" is the Cross-Join operator
    T_Effic_10 AS T_C
WHERE Switch( T_A.Num = 347
              , Switch( T_B.Num = 678
                      , Switch(T_C.Num = 7, True, True, False)
                      , True, False)
              , True, False) ;

```

If running this Query in your computer takes excessive time and you want to abort it, you may press “**Ctrl-Pause/Interr**” (click *J.11.18*).

The reason for taking so much time is that the **Cross-Join** operation is **first** producing a record-list with 10,000,000 records, and only **afterwards** the system is looking into these **very many** records **one by one**, to find which ones satisfy the “**WHERE**” **Boolean** expression. I have intentionally used the “**Switch()**” function in the “**WHERE**” **Boolean** expression to prevent the usage indexing thus forcing to process the records one by one.

However, if you replace the most restrictive “**WHERE**” **Boolean** expressions in inner Queries (even preventing the usage of indexing), you get the following Query<sup>150</sup> that

<sup>149</sup> This is the Query “**K\_Efficiency\_Select\_Queries\_1**” from file “**Company\_Database.accdb**”.

<sup>150</sup> This is the Query “**K\_Efficiency\_Select\_Queries\_2**” from file “**Company\_Database.accdb**”.

produces the same result and runs in **less than a second**:

```

SELECT T_A.Num, T_B.Num, T_C.Num
FROM
  (
    SELECT Num
    FROM T_Effic_1000
    WHERE Switch(Num = 347, True, True, False)
  ) AS T_A
, -- "Comma" is the Cross-Join operator
  (
    SELECT Num
    FROM T_Effic_1000
    WHERE Switch(Num = 678, True, True, False)
  ) AS T_B
, -- "Comma" is the Cross-Join operator
  (
    SELECT Num
    FROM T_Effic_10
    WHERE Switch(Num = 7, True, True, False)
  ) AS T_C ;

```

The reason is that in this case we produce **first** three record-lists, each with only one record, and **afterwards** we do the **Cross-Join** among the three records, producing just one **output** record. Even though we have also prevented the usage of indexing, it is **much faster** to search one by one over 2,100 records (the **total** number of records in the **three** input record-lists), than over 10,000,000 records (the result of the **Cross-Join** over the three Tables).

Notice that both Queries above bring from the database server **the same record-lists**, and therefore, network transfer **is not the cause** of the big performance difference. If you want to know more about performance and network transfer, you may click “[K.3.15 What is the delay of network access to a database?](#)”.

The same we have just seen for **Join** operations applies also to **Union** operations and **Select** operations: the **smaller** the record-lists of the **operands**, the **faster** they are processed.

In summary, you should always try to use the most restrictive “**WHERE**” and “**HAVING**” *Boolean* expressions.

### K.7.2.3 Why should I mainly use comparison and logical operators in my “WHERE” and “HAVING” expressions?

Because **Comparison** and **Logical** operators combined with **indexing** cause a **much faster processing** than if you use other operators and/or functions. You should therefore try to use **only** Comparison and Logical operators in your “**WHERE**” and “**HAVING**” *Boolean* expressions.

If you use the “**IN**” operator, the **Select** operation will run slower. If you use functions, or much worse, user-defined functions, this makes your indexes useless, and the Query processing may be **much** slower.

This is the same advice I am giving for your “**ON**” *Boolean* expressions.

Let me show the difference using the same example<sup>151</sup> from K.7.2.2. The following Query

<sup>151</sup> This is the Query “K\_Efficiency\_Select\_Queries\_1” from file “Company\_Database.accdb”.

takes **FOURTY FIVE seconds** to run!

```

SELECT T_A.Num, T_B.Num, T_C.Num
FROM
  T_Effic_1000 AS T_A
, -- "Comma" is the Cross-Join operator
  T_Effic_1000 AS T_B
, -- "Comma" is the Cross-Join operator
  T_Effic_10 AS T_C
WHERE Switch( T_A.Num = 347
            , Switch( T_B.Num = 678
                    , Switch(T_C.Num = 7, True, True, False)
                    , True, False)
            , True, False) ;

```

If running this Query in your computer takes excessive time and you want to abort it, you may press “**Ctrl-Pause/Interr**” (click *J.11.18*).

If you replace the “**Switch()**” function in the Query above by an expression composed of Comparison operators combined with Logical operators, you get the following Query<sup>152</sup> that produces the same result, and runs in **less than a second**, because **indexing** can be used:

```

SELECT T_A.Num, T_B.Num, T_C.Num
FROM
  T_Effic_1000 AS T_A
, -- "Comma" is the Cross-Join operator
  T_Effic_1000 AS T_B
, -- "Comma" is the Cross-Join operator
  T_Effic_10 AS T_C
WHERE (T_A.Num = 347) AND (T_B.Num = 678) AND (T_C.Num = 7) ;

```

Even though this Query is very efficient, it is **not the advisable way** to write this SQL operation, because it contains a large **Cross-Join**. The advisable way to write this operation is using a **Select** operation over the Tables that contain the most restrictive possible “**WHERE**” **Boolean** expression (click *K.7.2.2*).

### **K.7.3 How do I design faster Select operations over Tables?**

In addition to the general recommendations from the previous section “*K.7.2 How do I design faster Select operations over Queries?*”, when writing a **Select** operation over a **Table**, it is important that you take into account the following **two additional** efficiency considerations:

- “*K.7.3.1 Why should I avoid bringing unnecessary data from Tables?*”
- “*K.7.3.2 Why should I consider using uncorrelated Subqueries and/or domain aggregate functions in “WHERE” expressions over Tables?*”

#### **K.7.3.1 Why should I avoid bringing unnecessary data from Tables?**

Because the output record-list of each **Select** operation over a Table is usually brought, **over the network**<sup>153</sup>, from the database server to your computer. The **smaller** the record-list, the **faster** the network data transfer will be (click *K.3.15*). In summary, **avoid bringing unnecessary data** (neither records nor fields) from your Tables by carefully

<sup>152</sup> This is the Query “**K\_Efficiency\_Select\_Queries\_3**” from file “**Company\_Database.accdb**”.

<sup>153</sup> Unless the database Tables are stored in your local disk drive, which is quite unusual.

writing your “**WHERE**” expressions.

You can unwillingly bring **unnecessary** data in two ways:

- Bringing records and/or fields that are **not needed** for the Query processing.
- Bringing records and/or fields that **are needed** for the Query processing, but that are **brought multiple** times.

I explain this in more detail with a couple examples over the example Table “**T\_Effic\_3000**” (described in K.7).

The following Query<sup>154</sup> is **not efficient**, because it brings over the network **999 unnecessary** records, and also brings the **unnecessary** field “**Strng**”:

```
SELECT Num+78 AS Result
FROM
  (SELECT Num, Strng FROM T_Effic_3000)
WHERE Num = 456 ;
```

You can instead use the following Query, which is much **faster**, because it only brings over the network **the one record that is needed**, and **the one field that is needed**:

```
SELECT Num+78 AS Result
FROM T_Effic_3000
WHERE Num = 456 ;
```

Another example: a given Query containing the following **two Select** operations is **not efficient**, because it brings 600 fields **twice** over the network. The 600 fields brought twice correspond to field “**Num**” with values from “**200**” to “**799**”.

```
SELECT Num, Strng FROM T_Effic_3000 WHERE Num BETWEEN 100 AND 799
```

and

```
SELECT Num FROM T_Effic_3000 WHERE Num BETWEEN 200 AND 899
```

The Query optimizer will most likely fix this, but it is better practice to write your code efficiently just in case the Query optimizer is not as good as you thought. To avoid bringing data twice, you could replace the two **Select** operations above by:

```
SELECT Num, Strng
FROM
  (SELECT Num, Strng FROM T_Effic_3000 WHERE Num BETWEEN 100 AND 899)
WHERE Num BETWEEN 100 AND 799
```

and

```
SELECT Num
FROM
  (SELECT Num, Strng FROM T_Effic_3000 WHERE Num BETWEEN 100 AND 899)
WHERE Num BETWEEN 200 AND 899
```

which would be more efficient.

You probably have noticed that although these two **Select** operations are a great improvement over the first ones, they are still bringing some unnecessary data. Specifically, the values of field “**Strng**” from “**0799**” to “**0899**” are brought

---

<sup>154</sup> All the SQL operations in this section may be found in the Query “**K\_Efficiency\_Select\_Tables**” from file “**Company\_Database.accdb**”.

unnecessarily. You could remove this unnecessary data using the following third case of **Select** operations:

```
SELECT Num, Strng FROM T_Effic_3000 WHERE Num BETWEEN 100 AND 799
```

and

```
SELECT Num
FROM
  (SELECT Num, Strng FROM T_Effic_3000 WHERE Num BETWEEN 100 AND 799)
WHERE Num BETWEEN 200 AND 799
UNION
SELECT Num FROM T_Effic_3000 WHERE Num BETWEEN 800 AND 899
```

However, in order to gain some efficiency in bringing data over the network, we are making the SQL code more complex to understand and debug, and we are losing some efficiency increasing the processing time. Therefore, there is a **trade-off** between avoiding bringing unnecessary data and making your SQL code more complex and inefficient. The best design depends mainly on the size of record-lists handled. For large record-lists, it is more important avoiding bringing unnecessary data. For small record-lists, it is more important the code readability and code efficiency.

### **K.7.3.2 Why should I consider using uncorrelated Subqueries and/or domain aggregate functions in “WHERE” expressions over Tables?**

Because uncorrelated Subqueries (click [G.8.4](#)) and/or domain aggregate functions (click [G.6.2](#)) in a “**WHERE**” expression can be quite useful to avoid bringing unnecessary data from Tables (click [K.7.2.2](#)).

Imagine that you have a Table of projects “**T\_Projects**” with fields “**Begin**” and “**End**” indicating the begin and end date of each project. You also have a Table of invoices “**T\_Invoices**”, that has the field “**Project**” to identify to what project belongs the invoice. Imagine now that you want all the invoices of projects that are active along the current calendar year. The Query that would get them is:

```
SELECT Invoice_ID, Invoice_Amount, Invoice_Date, Project, ...
FROM T_Invoices
WHERE Project IN (SELECT Project
                  FROM T_Projects
                  WHERE Year(Date()) BETWEEN Year(Begin) AND Year(End))
```

I have highlighted the Subquery in turquoise.

As you may see, using uncorrelated Subqueries and/or domain aggregate functions in your “**WHERE**” *Boolean* expressions is very useful to minimize the size of record-lists brought from your Tables across the network.

However, in order to gain some efficiency in bringing data over the network, we are making the SQL code more complex to understand and debug, and we are losing some efficiency increasing the processing time. Therefore, there is a **trade-off** between avoiding bringing unnecessary data and making your SQL code more complex and inefficient. The best design depends mainly on the size of record-lists handled. For large record-lists, it is more important avoiding bringing unnecessary data. For small record-lists, it is more important the code readability and code efficiency.

### K.7.4 How do I design faster Union operations?

You should only use “**UNION**” if the **output** record-list **may have** duplicate records and you **do not want** duplicate records. This is exactly the same advice as the one for using the “**DISTINCT**” clause in **Select** operations (click *K.7.2.1*).

### K.7.5 How do I design faster Join operations?

I now provide a few recommendations for designing faster **Join** operations.

As a first advice, always use the most restrictive “**WHERE**” *Boolean* expressions in both **input** record-lists (click *K.7.2.2*).

You may also click:

- “*K.7.5.1 Why should I always restrict in inner Selects?*”
- “*K.7.5.2 Why should I mainly use comparison and logical operators in my “ON” expressions?*”

#### K.7.5.1 Why should I always restrict in inner Selects?

If you want to write a *Boolean* expression over the fields of an **input** record-list, **always** write it in the “**WHERE**” clause of **that** **input** record-list, and **never** write it in the “**ON**” clause, nor in the “**WHERE**” clause of the enclosing **Select** operation.

Let me show the difference using an example over the Tables “**T\_Effic\_**” (described in *K.7*). The following Query<sup>155</sup> takes in my laptop **THIRTY seconds** to run:

```
SELECT T_A.Num, T_B.Num
FROM
  T_Effic_3000 AS T_A
INNER JOIN
  T_Effic_3000 AS T_B
ON Switch(T_A.Num = T_B.Num, True, True, False)
WHERE Switch( T_A.Num = 1234, Switch(T_B.Num = 1234, True, True, False)
, True, False );
```

If running this Query in your computer takes excessive time and you want to abort it, you may press “**Ctrl-Pause/Interr**” (click *J.11.18*).

If you replace the outermost “**WHERE**” clause in the Query above by two innermost “**WHERE**” clauses, you get the following Query<sup>156</sup> that produces the same result and runs

<sup>155</sup> This is the Query “**K\_Efficiency\_Join\_innermost\_1**” from file “**Company\_Database.accdb**”.

<sup>156</sup> This is the Query “**K\_Efficiency\_Join\_innermost\_2**” from file “**Company\_Database.accdb**”.

in less than a second:

```
SELECT T_A.Num, T_B.Num
FROM
  (
    SELECT Num
    FROM T_Effic_3000
    WHERE Switch(Num = 1234, True, True, False)
  ) AS T_A
INNER JOIN
  (
    SELECT Num
    FROM T_Effic_3000
    WHERE Switch(Num = 1234, True, True, False)
  ) AS T_B
ON Switch(T_A.Num = T_B.Num, True, True, False) ;
```

### K.7.5.2 Why should I mainly use comparison and logical operators in my “ON” expressions?

Because **Comparison** and **Logical** operators combined with **indexing** cause a **much faster processing than** if you use other operators and/or functions. You should therefore try to use **only** Comparison and Logical operators in your “**ON**” *Boolean* expressions.

If you use the “**IN**” operator, the **Join** operation will run slower. If you use functions, or much worse, user-defined functions, this makes your indexes useless, and the Query processing may be **much** slower.

This is the same advice I am giving for your *Boolean* expressions in your “**WHERE**” and “**HAVING**” clauses.

Let me show the difference using an example over the Tables “**T\_Effic\_**” (described in K.7). The following Query<sup>157</sup> takes in my laptop **TWENTY seconds** to run!!

```
SELECT T_A.Num, T_B.Num
FROM
  T_Effic_3000 AS T_A
INNER JOIN
  T_Effic_3000 AS T_B
ON Switch(T_A.Num = T_B.Num, True, True, False)
ORDER BY T_A.Num ;
```

If running this Query in your computer takes excessive time and you want to abort it, you may press “**Ctrl-Pause/Interr**” (click *J.11.18*).

If you replace the “**Switch()**” function in the Query above by a Comparison operator, you get the following Query<sup>158</sup> that produces the same result and runs in **less than a second**, because **indexing** can be used:

```
SELECT T_A.Num, T_B.Num
FROM
  T_Effic_3000 AS T_A
INNER JOIN
  T_Effic_3000 AS T_B
ON T_A.Num = T_B.Num
ORDER BY T_A.Num ;
```

<sup>157</sup> This is the Query “**K\_Efficiency\_Join\_on\_exp\_1**” from file “**Company\_Database.accdb**”.

<sup>158</sup> This is the Query “**K\_Efficiency\_Join\_on\_exp\_2**” from file “**Company\_Database.accdb**”.

### Why should I use “INNER JOIN” instead of Cross-Join plus “WHERE”?

Because it will produce faster Queries. Even though the Query optimizer will most likely make a **Cross-Join** plus a “WHERE” clause as fast as an “INNER JOIN”, using a **Cross-Join** plus a “WHERE” clause is still a bad programming practice that should be avoided. The following Query<sup>159</sup> is an example of a **Cross-Join** plus a “WHERE” clause:

```
SELECT T_A.Num, T_B.Num
FROM
  T_Effic_3000 AS T_A
,
  T_Effic_3000 AS T_B
WHERE Switch(T_A.Num > Log(T_B.Num+100), True, True, False)
ORDER BY T_A.Num ;
```

And the following is an example of a Query that produces **exactly the same result**, but coding it with an “INNER JOIN”, which is **the advisable way to do it**:

```
SELECT T_A.Num, T_B.Num
FROM
  T_Effic_3000 AS T_A
INNER JOIN
  T_Effic_3000 AS T_B
ON Switch(T_A.Num > Log(T_B.Num+100), True, True, False)
ORDER BY T_A.Num ;
```

### K.7.6 How do I design a faster Select-group by when I have bound values in all my records?

Because when you have **two or more fields** that have **bound values**, and you want to do a “GROUP BY” over them, it is more efficient to do the “GROUP BY” over **only one** of the fields, and use “**First()**” over each of the other ones. You may click “*F.7.18.3 What are the “First()*” and “*Last()*” SQL aggregate functions?”.

Imagine that you have records that have the month name, the first day of the month, the last day of the month, and for each month you have several records each with the daily number of cars across a bridge. The field names could be “**Month\_name**”, “**Start\_Month**”, “**End\_Month**” and “**Daily\_Cars**”. If you wanted to get the **monthly average number** of cars across the bridge, you could write:

```
SELECT Month_name, Avg(Daily_Cars)
      * (End_Month-Start_Month+1) AS Monthly_Cars
FROM T_Bridge_Cars_per_Day
GROUP BY Month_name, Start_Month, End_Month
```

However, it is more efficient to write:

```
SELECT Month_name, Avg(Daily_Cars)
      * (First(End_month)-First(Start_Month)+1) AS Monthly_Cars
FROM T_Bridge_Cars_per_Day
GROUP BY Month_name
```

The reason is that the first SQL operation has to group all the records on the values of **three** fields, while the second one groups them on **only one** field, and the latter requires less processing time.

---

<sup>159</sup> This is the Query “K\_Efficiency\_Join\_inner\_1” from file “Company\_Database.accdb”.











- “K.9.3 How do I write a VBA function that reads database records?”
- “K.9.4 How do I write a VBA function that requests a parameter to the user?”
- “K.9.5 How do I test my user-defined VBA functions?”
- “K.9.6 How do I write VBA Subroutines associated to Form Events?”
- “K.9.7 How do I close the VBA editor?”

### **K.9.1 How do I create VBA modules and open the VBA editor?**

MS-Access allows you to write user-defined VBA functions that can be used in Table menus, Queries and Forms, providing a very powerful and flexible processing environment.

Click on the “**Create**” Ribbon name, and then on the **Module** “” icon, in the top right corner of the “**Macros & Code**” Ribbon group. This will create a new VBA module, and open the VBA editor. The VBA editor will be opened in a **different window** to the MS-Access window. Click on the **save** “” icon from the VBA editor, and you will be prompted to enter the name of the module. Type-in a name and either press the “**Enter**” key or click on the “**Yes**” button. You have now created a VBA module in your database to store you user-defined functions. Each VBA module is placed as an object in the “**Navigation Pane**”, under the category “**Modules**”.

My advice is you write the following text:


```
Option Compare Database
Option Explicit
```

at the top of your module.

The option “**Compare Database**” will make string comparison be based on the same ordering of your MS-Access database. This is extremely useful to avoid puzzling results arising from having **two different** ordering criteria in your VBA and SQL code.

The option “**Explicit**” obliges you to write a variable declaration of all the variables you use in your VBA code. This may sound like more work, but it is **much, much, safer** and it is certainly worth the extra work. It has an added advantage that your VBA code will **run faster**.

Below these two options you may write one by one all your user-defined VBA functions.

Once you are done editing your VBA module, **save** your changes by clicking on the **save** “” icon, and **close** the VBA editor by clicking on the **close** icon “**X**” in the top right corner of its window.

You may open a VBA module with the “**Navigation Pane**” (click *B.4.1.3*). This will open the VBA editor window, with your VBA module opened inside it.

You can invoke all your user-defined VBA functions both from other of your user-defined VBA functions and from any expression in your SQL code.


After you have completed some modifications in your VBA code, I strongly advise that you to click on “**Debug**” (on the top window frame) and then click on “**Compile VBA\_project\_name**” from the pop-up menu (where *VBA\_project\_name* is the name of your VBA **Module**, most frequently “*Database1*”). This will cause the VBA editor to

**compile** your code and will show you any error(s) it may find, so you can **fix** them.

The editor also has the following commands to **debug** your VBA code:

- **Introduce/clear an execution breakpoint**


You may introduce/clear a breakpoint in either of the following ways:

- Double-click on the gray vertical margin on the left of your VBA code, aligned with the line of code where you want to introduce/clear the breakpoint.
- Press the “**F9**” key. The breakpoint will be introduced in the line of code where the text cursor (not the mouse pointer) is currently placed.
- Click on “**T**oggle Breakpoint ” from the “**D**ebug” pop-up menu. The breakpoint will be introduced in the line of code where the text cursor (not the mouse pointer) is currently placed.


- **Advance the program flow one step**

Either press the “**F8**” key or click on “**S**tep Into ” from the “**D**ebug” pop-up menu.

- **Advance the program flow one step, without entering procedures**

Either simultaneously press the “**S**hift” and “**F8**” keys, or rather, click on “**S**tep Over ” from the “**D**ebug” pop-up menu.

- **Advance the program flow out of the current procedure**

Either press “**C**trl-**S**hift-**F8**” (i.e., simultaneously press the “**C**trl” and “**S**hift” keys, and without releasing them, press the “**F8**” key), or rather, click on “**S**tep Out ” from the “**D**ebug” pop-up menu.

- **Advance the program flow up to the line where the cursor (not the mouse pointer) is currently placed**

Either press “**C**trl-**F8**” (i.e., press the “**C**trl” key, and without releasing it, press the “**F8**” key), or rather, click on “**R**un to Cursor ” from the “**D**ebug” pop-up menu.

- **Clear ALL the execution breakpoints**

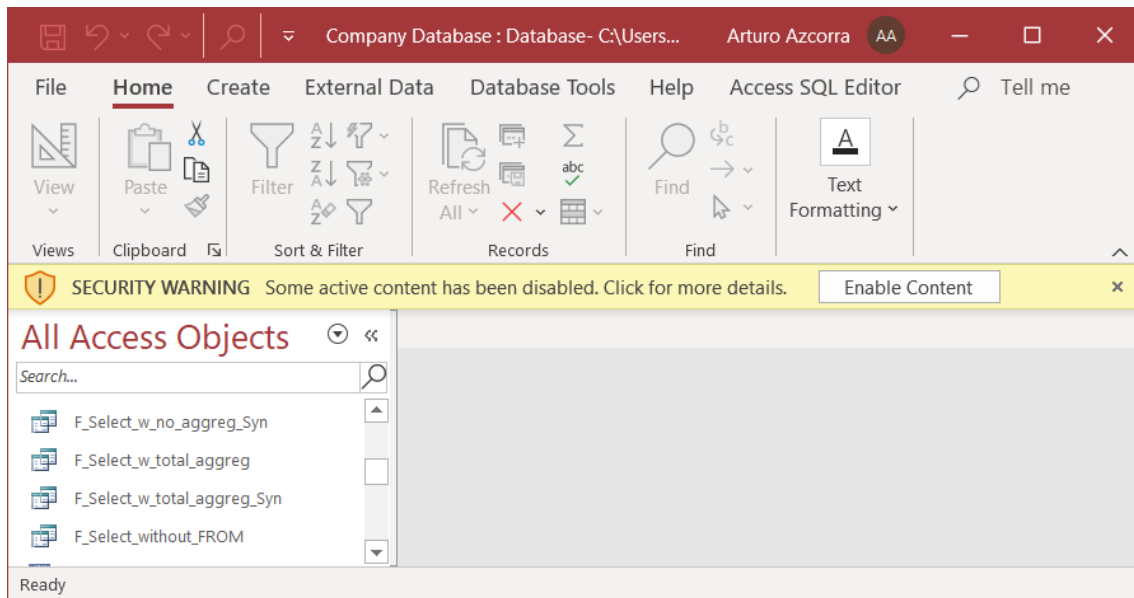
Either press “**C**trl-**S**hift-**F9**” (i.e., simultaneously press the “**C**trl” and “**S**hift” keys, and without releasing them, press the “**F9**” key), or rather, click on “**C**lear All Breakpoints” from the “**D**ebug” pop-up menu.

Breakpoints are shown as a dark red circle “●” placed, at the beginning of the corresponding code line, in the gray vertical margin on the left of your VBA code. Also, the whole code line is **highlighted** in **dark red** color.

When the program execution is stopped you may see the **content of any variable** by placing the mouse pointer over it (a box showing the value will pop-up).

If you write user-defined VBA functions or macros, when you open an MS-Access file

you may get the **yellow** warning message shown in the following screenshot:



You should always click on “**Enable Content**” and continue working normally. This is the standard MS-Office security policy, and you will get the same warning when you open a file with user defined functions/macros in any MS-Office application.

### **K.9.2 How do I write a VBA function that inserts/updates/deletes Table’s records?**

Using the VBA built-in method “**Execute**” over the built-in database object “**CurrentDb**”.

You first assign to a VBA text string variable the SQL **Insert**, **Update** or **Delete** operation (click *F.13*) that you want. For example:

```
Dim SQL_command As String
SQL_command = "DELETE FROM T_Insert_Delete WHERE Capital=""Brussels" ; "
```

Notice that in order to include **one double quote** character inside a text string you have to **write two** double quote characters. Notice you **cannot** use the single quote character as string delimiter because in VBA it is interpreted as a comment delimiter.

The SQL operation contained in the *String* variable “SQL\_command” above will output all the records from the Table “T\_Capital\_Rainfall” where the value of “Capital” is equal to “Brussels”.

To run the SQL code contained in the *String* variable “SQL\_command” you invoke the method “**Execute**” from the object “**CurrentDb**”. Continuing with the example, the resulting VBA code is:

```
Dim SQL_command As String
SQL_command = "DELETE FROM T_Insert_Delete WHERE Capital=""Brussels" ; "
CurrentDb.Execute SQL_command
```

And this is all you need to run the SQL code that **Inserts**, **Updates** or **Deletes** records into the database.

In case that you want that the SQL code depends on a VBA variable, you just need to replace the corresponding text in the SQL command string. Continuing with the example, imagine that you want “Brussels” to be the content of a VBA *String* variable called “MyCity”. Then, the resulting VBA code would be:

```
Dim SQL_command As String
SQL_command = "DELETE FROM T_Insert_Delete WHERE Capital="""
SQL_command = SQL_command & MyCity & """" ; "
CurrentDb.Execute SQL_command
```

Notice that in order to include **one double quote** character inside a text string you have to **write two** double quote characters, and the third double quote that you see is the one that marks the end of the text string. Notice you cannot use the single quote character as string delimiter because in VBA it is interpreted as a comment delimiter.

### **K.9.3 How do I write a VBA function that reads database records?**

Using the VBA built-in method “**OpenRecordset**” over the built-in database object “**CurrentDb**”.

You first assign to a VBA text string variable the SQL operation that produces the output record-list that you want to read from the database. For example:

```
Dim SQL_command As String
SQL_command = "SELECT Cal_Year, Quart, Quart_Rainfall FROM
T_Capital_Rainfall WHERE Capital = ""Washington"";"
```

Notice that in order to include **one double quote** character inside a text string you have to **write two** double quote characters. Notice you **cannot** use the single quote character as string delimiter because in VBA it is interpreted as a comment delimiter.

The SQL operation contained in the *String* variable “SQL\_command” above will output all the records from the Table “T\_Capital\_Rainfall” where the value of “Capital” is equal to “Washington”.

To run the SQL code contained in the *String* variable “SQL\_command” above, you define a variable to store the resulting output record-list (data type *Recordset*) and you invoke the method “**OpenRecordset**” from the object “**CurrentDb**”. Continuing with the example, the resulting VBA code is:

```
Dim SQL_command As String, Output_records As Recordset
SQL_command = "SELECT Cal_Year, Quart, Quart_Rainfall FROM
T_Capital_Rainfall WHERE Capital = ""Washington"";"
Output_Records = CurrentDb.OpenRecordset(SQL_command)
```

You now have the resulting output record-list accessible through the variable “**Output\_Records**”, of VBA data type *Recordset*. You can apply the following methods over this variable:

- **MoveFirst**: moves the record pointer to the **first** record of the *Recordset* variable.
- **MoveLast**: moves the record pointer to the **last** record of the *Recordset* variable.
- **MoveNext**: moves the record pointer to the **next** record of the *Recordset* variable.
- **MovePrevious**: moves the record pointer to the **previous** record of the

*Recordset* variable.

- **Move**: moves the record pointer **n** positions **forward** of the *Recordset* variable.
- **Edit**: opens the **current** record to **modify** its field values.
- **Update**: **updates in the database** the **current** record with the field values that have been modified with the “**Edit**” method.
- **EOF**: returns *True* if there is the **current** record is **Null**. This tells you that you have reached the **end** of the record-list.

A usual way to handle a record-list is by doing a while-loop until the “**EOF**” method returns *True*, in which case the full record-list has been processed.

#### **K.9.4 How do I write a VBA function that requests a parameter to the user?**

Invoking the built-in function “**InputBox()**” that returns the **text string** typed-in by the user. The function “**InputBox()**” pops-up a dialogue-box on the screen and returns the **text string** that has been typed-in by the user into the dialogue-box. The way to write (syntax) an invocation of “**InputBox()**” is:

```
String_variable = InputBox(prompt [[, title ], default ], Other ] )
```

Optional arguments are enclosed between square brackets.

Argument “**prompt**” is a **String**. It is the text string that will be shown to the user in the dialogue-box that requests him to enter data. The maximum length of the “**prompt**” *String* is around 1,024 characters, depending on the width of the characters used. You may split the text that is shown in several lines by including line feed “**Chr(10)**” inside the text string of “**prompt**”.

**Optional** argument “**title**” is the text string shown as **the title** of the dialogue-box. If it is not used, then the application name will be shown.

**Optional** argument “**default**” is the default text string returned by the function “**InputBox()**” in case the user does not provide any value.

Other **optional** arguments allow you to select the position of the dialogue-box and other side issues, that you can check on-line in case you need to use them.

#### **K.9.5 How do I test my user-defined VBA functions?**

My advice is that you test each of your user-defined VBA function using a specific test-Query. The advantages of doing this are the following:

- Queries are quite good for generating different value combinations to test your user-defined functions.
- Queries are quite good for visually checking the results of the function over different value combinations.
- Queries can be easily saved in a MS-Access file to use them the next time you do any modification on your user-defined function that does not modify its intended functionality (e.g., making it more efficient). Even if the VBA modification implies a change in the functionality of your user-defined VBA function, it is usually worth the



work to adjust the test Query.

Linking with the last bullet point, I strongly advise you to run the test-Query associated to a given user-defined function **always** after having made **any** modification on the VBA code of the user-defined function. Remember it is **extremely easy to introduce an error** in your VBA code, even if you think that the change you made cannot produce any side effect.

My advice to write your test-Queries is the following.

Each **test case** is written as a **Select** operation like the following one:

```
SELECT value_1 AS arg_1, value_2 AS arg_2, ..., value_n AS arg_n
      , Func_name(arg_1, arg_2, ..., arg_n) AS Result
      , Iif(Result = Expected_value_1, "OK", "ERROR") AS CHECK
FROM T_Numbers WHERE Num = 1
```

The first line:

```
SELECT value_1 AS arg_1, value_2 AS arg_2, ..., value_n AS arg_n
```

is the **value combination** of arguments that you want to use in this **test case**. This first line is **different** in each test case, because in each test case you will want to use a different combination of values as arguments of your VBA user-defined function.

The second line:

```
      , Func_name(arg_1, arg_2, ..., arg_n) AS Result
```

is the actual invocation of the VBA user-defined function. The value returned by the user-defined function is assigned the value name “**Result**”, to be used in the expression in the third line. This second line is **the same** in all your test-cases.

The third line:

```
      , Iif(Result = Expected_value_1, "OK", "ERROR") AS CHECK
```

checks if the value returned by the function (called “**Result**”), is what you expect for this test case. The way to check this is by using an “**Iif()**” function where the **Boolean** expression compares the **actual result** “**Result**” with your **expected result** “**Expected\_value**” for the arguments contained in the first line. If the **Boolean** expression is **True**, the “**Iif()**” returns “**OK**” and it otherwise returns “**ERROR**”. This third line is **the same** in all your test-cases, except “**Expected\_value**” that will be different.

The fourth line:

```
FROM T_Numbers WHERE Num = 1
```

is only needed to produce one record with the values indicated after the “**SELECT**” clause.

For the case of fractional results, it may be convenient to get an approximate result instead of an exact result, to avoid problems with decimal-binary conversion. If this is what you want, then you may replace the third line by the following one:

```
      , Iif(abs(Expected_result - Result) < 1e-6, "OK", "ERROR") AS CHECK
```

As you may see, in the example above we are getting the same result with a precision of  $10^{-6}$ , but of course you may change this precision level by the one required in each particular case.

You build as many test cases as you want using the **Select** operation template that I have indicated, and you connect them with the “**UNION ALL**” SQL operator. The resulting

test-Query would be:

```

SELECT value_1 AS arg_1, value_2 AS arg_2, ..., value_n AS arg_n
      , Func_name(arg_1, arg_2, ..., arg_n) AS Result
      , Iif(Result = Expected_value_1, "OK", "ERROR") AS CHECK
FROM T_Numbers WHERE Num = 1
UNION ALL
...
UNION ALL
SELECT value_x AS arg_1, value_y AS arg_2, ..., value_z AS arg_n
      , Func_name(arg_1, arg_2, ..., arg_n) AS Result
      , Iif(Result = Expected_value_k, "OK", "ERROR") AS CHECK
FROM T_Numbers WHERE Num = 1

```

Imagine that you want to test your user-defined function “**Weekend\_Days()**” that returns the number of Saturdays or Sundays between two dates, both included. Following my advice, you could write the following Query<sup>160</sup> code:

```

SELECT #1/1/2013# AS Begin, #5/1/2013# AS End
      , Weekend_days(Begin, End) AS Result
      , Iif(Result=34, "OK", "ERROR") AS CHECK
FROM T_Numbers WHERE Num = 1
UNION ALL
SELECT #1/1/2013# AS Begin, #5/1/2014# AS End
      , Weekend_days(Begin, End) AS Result
      , Iif(Result=138, "OK", "ERROR") AS CHECK
FROM T_Numbers WHERE Num = 1

```

If you run the Query above, you would get the following result:

K_Test_functions			
Begin	End	Result	CHECK
01/01/2013	01/05/2013	34	OK
01/01/2013	01/05/2014	138	OK

Obviously, for a realistic test-Query you would write many more test-cases than only just two, as in the example above. If the number of test-cases is large, it may be convenient to enclose all of them in a **Select** with the “**WHERE**” clause:

```
WHERE Check = "ERROR"
```

in order to show only the wrong results, thus avoiding possible errors from visually looking for them.

The Table “T\_Numbers” used in the example above is an auxiliary Table with only one field (named “Num”) that just contains integer numbers (click [K.2.2](#)).

### **K.9.6 How do I write VBA Subroutines associated to Form Events?**

If you want to do this, you may click “[D.10.4 How do I configure VBA action code associated to Form Events?](#)”.

### **K.9.7 How do I close the VBA editor?**

If at this moment you are reading this **Lightning Guide linearly**, click to read “[D.10.4.5 How do I close the VBA editor?](#)” and then return here (you return by

<sup>160</sup> This is the Query “K\_Test\_functions” from file “Company\_Database.accdb”.

simultaneously pressing the “Alt” and “←” keys).

## **K.10 What elements/concepts are explained in various places?**

Some elements affect database design in several ways and for this reason appear explained in different places of this Lightning Guide. To simplify your looking for the correct place, you may click on the topic you want from the following list:

- “K.10.1 Where are names and identifiers explained?”
- “K.10.2 Where are drop-down menus explained?”
- “K.10.3 Where are duplicates explained?”
- “K.10.4 Where are indexes explained?”
- “K.10.5 Where are Key fields explained?”
- “K.10.6 Where are Relationships explained?”
- “K.10.7 Where is Query results ordering explained?”
- “□ “H.2.3 How do I change the sorting of rows in a Table/Query/Form?””
- “K.6.12 How do I get the exact record ordering I want?”
- Where are data types explained?”
- “K.10.9 Where are zero-length and invisible strings explained?”
- “K.10.10 Where are exception-values explained?”
- “K.10.11 Where are aggregate functions explained?”
- “K.10.12 Where is remote database access explained?”
- “K.10.13 Where is VBA code explained?”
- “K.10.14 Where are Nulls explained?”

### **K.10.1 Where are names and identifiers explained?**

- “C.2.1 What is an object and a name?”
- “C.2.2 What is a qualified field name?”
- “D.2 How do I carefully assign good names from the very beginning?”

### **K.10.2 Where are drop-down menus explained?**

- “B.11 Can I use a drop-down/expression menu even if its icon is not shown?”
- “D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?”
- “K.1.7 Why should I configure drop-down menus to enter data?”
- “K.1.8 What are good practices in configuring my drop-down menus?”
- “K.1.9 How do I configure a drop-down menu in a Date/Time field?”

- *“L.4.2.5 How do I fix a value rejected because it is not in the list?”*
- *“L.5.10 How do I fix a value not in the drop-down menu in a Table/Form field?”*

### **K.10.3 Where are duplicates explained?**

- *“C.7 What are duplicate records and duplicate field values?”*
- *“C.8.3.2 What are indexes with duplicate values and without duplicate values?”*
- *“C.8.3.5 What is an index without duplicate values and without Nulls?”*
- *“C.9 How do I prevent duplicate field values and duplicate records?”*
- *“F.7.8 What is the “DISTINCTROW” clause of a Select?”*
- *“F.7.11 What is the “DISTINCT” clause of a Select?”*
- *“F.9.1 What are the Union operators?”*
- *“F.9.4.2 What are the output records of a Union?”*
- *“J.14.6 What effects does Null cause in SQL operators that remove duplicate records?”*
- *“K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?”*
- *“K.4.6 When should I remove duplicate records?”*
- *“L.2.6 How do I fix “Key/Index with duplicate values” when saving my Table design?”*
- *“L.4.3.2 How do I fix erroneous records because of duplicate values?”*

### **K.10.4 Where are indexes explained?**

- *“B.6.2 How do I manage Table indexes in “Design View”?”*
- *“C.8 What is indexing?”*
- *“D.5 How do I configure a Table field validation rule, indexing, and other properties?”*
- *“D.7 How do I add simple and/or composite index(es) to a Table?”*
- *“I.5.6 What are the side effects of modifying the indexes or the Key fields of a Table?”*
- *“K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?”*
- *“K.2.4 What are the interactions between Relationships and “PrimaryKey”, “Required” and Table indexes?”*
- *“L.2.6 How do I fix “Key/Index with duplicate values” when saving my Table design?”*

### **K.10.5 Where are Key fields explained?**

- *“C.10 What are the Table Key(s) and how should I handle them?”*

- “D.6 How do I configure the Primary Key field(s) of a Table?”
- “D.7 How do I add simple and/or composite index(es) to a Table?”
- “I.5.6 What are the side effects of modifying the indexes or the Key fields of a Table?”
- “K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?”
- “K.2.4 What are the interactions between Relationships and “PrimaryKey”, “Required” and Table indexes?”
- “L.2.6 How do I fix “Key/Index with duplicate values” when saving my Table design?”
- “L.2.7 How do I fix “...primary key cannot contain a Null...” when saving my Table design?”
- “L.2.10 How do I fix “There is no primary key...” when saving my Table design?”

### **K.10.6 Where are Relationships explained?**

- “A.9 How do I create a Relationship in my first database?”
- “B.10 What is the “Relationships” pane?”
- “C.11 What is a Relationship?”
- “D.9 How do I create and configure my Table Relationships?”
- “I.8 What are the side effects of modifying my Relationships, Forms and/or Reports?”
- “K.2.4 What are the interactions between Relationships and “PrimaryKey”, “Required” and Table indexes?”
- “L.3 How do I fix errors in my Relationship configuration?”
- “L.4.3.3 How do I fix a slave record without a master record?”

### **K.10.7 Where is Query results ordering explained?**

- “F.7.12 How do I use “ORDER BY” to order the output records of a Select?”
- “F.10.8 What is the “ORDER BY” clause of a Transform?”
- “H.2.3 How do I change the sorting of rows in a Table/Query/Form?”
- “K.6.12 How do I get the exact record ordering I want?”

### **K.10.8 Where are data types explained?**

- “D.4 How do I configure a Table field data type and size?”
- “F.7.18.7 What is a summary and grouping of aggregate functions?”.
- “G.2 How do I manage VBA data types and Table field types-sizes?”
- “G.3 What is the data type returned by an expression?”
- “G.5 How do I use value operators in an expression?”

- “G.9 How are numeric-like values internally represented and processed?”
- “J.16 What data type bugs can I get?”

### **K.10.9 Where are zero-length and invisible strings explained?**

- “D.5.2.2 What is the “Allow Zero Length” Table field property?”
- “G.4.2 How do I write String constants?”
- “G.5.2 What are the Text string operators?”
- “I.4.4.2 What are the side effects of changing the “Required”, “Allow zero length”, “Validation rule” or “Indexing” properties of a Table field?”
- “L.4.2.3 How do I fix violating “Allow Zero Length=No”?”
- “L.5.13 How do I fix an apparent zero-length string in a Table/Form field configured as “Allow Zero Length=No”?”
- “L.7 How do I fix errors with Short Text or String fields?”

### **K.10.10 Where are exception-values explained?**

- “F.7.18 What is an SQL aggregate function?”
- “G.5 How do I use value operators in an expression?”
- “L.1 Why can I get an error/crash in a test-and-proven database?”
- “J.7.3 How do I debug the current uncommented SQL operation so it does not produce defective results?”
- “J.10 How do I fix a crash from a run-time error?”
- “J.11.10 How do I fix a Query producing “#Num!”, “#Div/0!”, “#Error”, “#Type!” or “#Func!” in a field?”
- “J.11.11 How do I fix a Query producing “#Invalid” or “#Deleted” in a field?”
- “J.11.13 How do I fix a Query producing “#Name?” in a field?”
- “J.14.7 What effect does Null cause in Union SQL operators?”
- “J.15 What exception-value bugs can I get?”

### **K.10.11 Where are aggregate functions explained?**

- “F.7.18 What is an SQL aggregate function?”
- “G.1.7 How domain aggregate functions depend on expression scopes?”
- “G.1.8 How SQL aggregate functions depend on expression scopes?”
- “G.6.2 How do I use domain aggregate functions in an expression?”
- “G.6.3 How do I use SQL aggregate functions in an expression?”
- “J.10.8 How do I fix the crash “Cannot have ... in aggregate argument.”?”
- “J.10.14 How do I fix a crash from my user-defined VBA functions?”

- “K.6.14 What are useful tricks with SQL aggregate functions?”
- “M.7 Domain Aggregate functions”
- “M.16 SQL aggregate functions”

### **K.10.12 Where is remote database access explained?**

- “D.13 How do I share a database, having multiple concurrent users?”
- “K.3 How do I structure and optimize a distributed database?”
- “L.5.15 How do I fix spontaneously changing Table/Form field values?”
- “L.6 How do I fix a Table/Form that I cannot open?”

### **K.10.13 Where is VBA code explained?**

- “C.2.8 What is a function?”
- “D.10.4 How do I configure VBA action code associated to Form Events?”
- “F.13.4 Can I write a VBA function that deletes, inserts or updates Table records?”
- “G.2 How do I manage VBA data types and Table field types-sizes?”
- “G.4.5 What are the restrictions when writing constants in VBA expressions?”
- “G.6.4 How do I use user-defined VBA functions in an expression?”
- “G.8.5 How do I use an SQL operation in a VBA variable assignment?”
- “I.7 What are the side effects of modifying my user-defined VBA functions?”
- “J.10.14 How do I fix a crash from my user-defined VBA functions?”
- “J.11.22 How do I fix my VBA functions comparing text strings case sensitive?”
- “J.14.9 What effects does Null cause as an argument of a VBA function?”
- “J.16.5 What data type bugs can I get with VBA functions?”
- “K.8.3 Why should I avoid using the Variant-Decimal VBA data type?”
- “K.8.4 How do Decimal data types work?”
- “K.9 How do I write my user-defined VBA functions and database Subroutines?”
- “L.8.11 How do I fix the VBA editor changing the value of the constants I write?”

### **K.10.14 Where are Nulls explained?**

- “C.6 What is a Null?”
- “C.7.1 What are duplicate records?”
- “C.8.3 What different types of indexes are there?”
- “C.10 What are the Table Key(s) and how should I handle them?”
- “D.5.1.7 What is the “Required” Table field property?”
- “G.5 How do I use value operators in an expression?”

- “F.7.18 What is an SQL aggregate function?”
- “F.8.7 What is the output record-list of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?”
- “F.8.8 What is the output record-list of a Full-Outer-Join?”
- “I.4.4.2 What are the side effects of changing the “Required”, “Allow zero length”, “Validation rule” or “Indexing” properties of a Table field?”
- “J.10.6 How do I fix the crash “Invalid use of Null.”?”
- “J.14 What Null-related bugs can I get?”
- “J.15 What exception-value bugs can I get?”
- “K.1.3 Why should I prevent Nulls in my Table fields?”
- “K.2.3 What are the interactions between Nulls, duplicates, indexing, and Key field(s)?”
- “K.2.4 What are the interactions between Relationships and “PrimaryKey”, “Required” and Table indexes?”
- “K.5 Why and how should I carefully handle Nulls in my Queries?”
- “L.2.1.1 How do I fix “Nulls in Required field” when saving my Table design?”
- “L.2.7 How do I fix “...primary key cannot contain a Null...” when saving my Table design?”
- “L.4.1.5 How do I fix a Table Short Text field configured as “Required=Yes” that accepts a Null?”
- “L.4.2.2 How do I fix trying to save Null in a “Required” field?”
- “L.5.11 How do I fix a Null in a Table/Form field configured as “Required=Yes”?”
- “L.5.12 How do I fix an apparent Null in a Table/Form Short Text field configured as “Required=Yes”?”



## PART L. FIXING DATABASE ERRORS

You may click:

- “[L.1 Why can I get an error/crash in a test-and-proven database?](#)”
- “[L.2 How do I fix errors with my Table/Form design?](#)”
- “[L.3 How do I fix errors in my Relationship configuration?](#)”
- “[L.4 How do I fix errors when entering, modifying or deleting records?](#)”
- “[L.5 How do I fix errors in Table/Form data?](#)”
- “[L.6 How do I fix a Table/Form that I cannot open?](#)”
- “[L.7 How do I fix errors with Short Text or String fields?](#)”
- “[L.8 How do I fix errors with the user interface?](#)”

### L.1 Why can I get an error/crash in a test-and-proven database?

When you are **starting** to build a database, you will **frequently** get error messages and crashes, that you will **fix** until the database seems to work properly and you can use it in production. You are clearly prepared to face this, and I am providing guidance in **Part J** and in this **Part L** about how to fix different problems you may encounter along your Query and database design.

However, when you have **extensively tested and proven** your database, and it has actually been in production for months, getting a severe error (a Query crash, a user-defined function crash, a clearly wrong result, ...) may cause you substantial **anxiety**. The first time I got a Query crash during normal operation of my database I was **in panic**, and I thought all the work I had invested in my database would now be useless.

If you get a Query **error/crash** during database operation you **should not worry**: this is absolutely **normal**, and it will **for sure** happen now and then.

Some examples of errors/crashes you may get (in increasing order of severity) are:

1. A Query result that is wrong (e.g., a date or number value that is not what should be).
2. A Query result that contains **exception-values** (e.g., “#Div/0!”, “Num!”, “#Error”, “#Type!”, “Func!”, “#Name?” or “#Invalid”).
3. A Query **crash** with an MS-Access error message indicating why the Query cannot be run.
4. A Query **crash** because a user-defined VBA function **crash**. The function **crash** can be reported with an MS-Access error message or by MS-Access opening the VBA editor/debugger.

When this happens and you get an error/crash in a **test-and-proven** Query, you can **fix** it by clicking “[J.1 How do I fix an error/crash in a test-and-proven Query?](#)”.

## L.2 How do I fix errors with my Table/Form design?

This chapter addresses errors and problems you may get with Table/Form design, in particular errors when trying to save the design of a **Table** or **Form**. Find the specific error you have among the following sections:

- “L.2.1 How do I fix data integrity errors when saving my Table design?”
- “L.2.2 How do I fix orphan Calculated fields when saving my Table design?”
- “L.2.3 How do I fix “Calculated column cannot be saved...” when saving my Table design?”
- “L.2.4 How do I fix “Could not find field...” when writing the expression of a Calculated field?”
- “L.2.5 How do I fix “Could not find field...” when saving my Table design?”
- “L.2.6 How do I fix “Key/Index with duplicate values” when saving my Table design?”
- “L.2.7 How do I fix “...primary key cannot contain a Null...” when saving my Table design?”
- “L.2.8 How do I fix “You can’t change the primary key” when saving my Table design?”
- “L.2.9 How do I fix “Cannot delete this index...” when saving my Table design?”
- “L.2.10 How do I fix “There is no primary key...” when saving my Table design?”

### L.2.1 How do I fix data integrity errors when saving my Table design?

If saving your Table design, you get the **warning** message:

**“Data integrity rules have been changed: existing data may not be valid for the new rules.**

*This process may take a long time. Do you want the existing data to be tested with the new rules?”*

this is because the changes you did in your Table design **may cause** data integrity errors. Data integrity errors arise if the Table data violates your Table design. The causes of these **possible** integrity errors are:

- The field(s) just configured as “**Required=Yes**” contain(s) **Null** in records **already existing** in the Table (click *D.5.1.7*, *D.8.1* and *I.4.4.2*).
- The **field validation rule**(s) just configured is(are) violated (i.e., it returns **False**) in records **already existing** in the Table (click *D.5.1.5* and *I.4.4.2*).
- The **record validation rule** just configured is violated (i.e., it returns **False**) in records **already existing** in the Table (click *D.8.1* and *I.5.7*).
- Field(s) just configured as “**Allow Zero Length=No**” contain(s) a **zero-length** string in records **already existing** in the Table (click *D.5.2.2* and *I.4.4.2*). You get the **warning** message above if you changed this, but even if you click “**Yes**” to the

question above, MS-Access will **not** check this. This may be an MS-Access **bug**.

- The new field type and/or field size just configured may cause Table data modification (click *D.4* and *I.4.4.I*), which in turn, may **also** cause one (or more) of the problems above.

The warning message at the beginning of this section is **not** notifying a data integrity error: it is a **warning** notifying the **possibility** of data integrity errors. Because the **possible** errors would be very harmful, you need to **handle this very carefully** just in case **errors actually exist**:

- You should click on “**Yes**” on the warning confirmation box, because otherwise MS-Access **will not check** your Table data for Table design violations. You **definitively want** to know if your Table data violates the Table design, and in case of errors, you want to fix them right away. If Table data that violates the Table design **remains undetected**, it will most likely create confusion, misunderstandings and errors in your database data and Query results.
- If you rather click on “**Cancel**”<sup>161</sup>, this will **cancel** the operation of **saving** your Table design changes, and the Table will remain open in “**Design View**”.
- If you rather click on “**No**”, your Table design changes will be **saved without checking if the Table data violates** them: as I indicated right above, this is **extremely bad**, so you have to be **very sure** that this is what you want before clicking on “**No**”.

If you clicked on “**Yes**”, MS-Access will **check** the Table data for possible violations of the **design changes you are trying to save**. This is, **the design changes** made since the **last** Table design was saved. Notice that MS-Access **will not** check for configuration violations of Table configuration that had been **previously saved**.

If **all the checks** by MS-Access are **correct**, it means that **all** Table data **complies** with the Table design changes being saved: the design changes will be saved, and the Table will be silently closed.

However, if MS-Access **finds a case of** Table data **violating** the design changes being saved, (e.g., a field just being set as “**Required=Yes**” having **Null**), MS-Access will show an integrity error message. There are **different types of data integrity violations**, and each of them has its specific integrity **error** message.

To fix this, my advice is that you sequentially take the following actions:

1. You should first **take careful note** of the **specific integrity error message(s)**, and also take note of the **specific design element(s)** shown in each of the integrity error message(s).
2. On the violation message confirmation buttons, if you are sure that you want your new configuration (e.g., “**Required=Yes**”), you should click on “**Yes**”: this will keep the configuration element and MS-Access **will continue checking** for **other** possible Table design violations.

If you rather click on “**No**”, this will **revert** the configuration element (e.g., a new or

---

<sup>161</sup> Closing the confirmation window clicking on its “**X**” icon is the same as clicking on “**Cancel**”.

modified field validation rule) that triggered the error, and MS-Access **will continue checking** for **other** possible design violations.

You should **almost never** click on “**Cancel**”<sup>162</sup>, because if you do, MS-Access **will stop checking** Table data for **other** possible design violations. Therefore, you **may have other** Table data (in addition to the one already reported up to now) that is violating your Table configuration, and it is **extremely bad** to ignore it.

3. If you clicked on “**Yes**” or “**No**”, MS-Access **will continue checking** the data. If MS-Access detects **another** case of Table data **violating** the design changes, MS-Access will show **another** violation warning message. You fix this by going back to action number **1** above.
4. You will have to repeat this cycle of actions **1**, **2** and **3** several times, as long as MS-Access keeps finding Table data violating the Table design configuration that you are trying to save. Once MS-Access does not find any more Table data violating the Table design (or you did click on “**Cancel**”, against my advice), the Table design changes will be saved unless **index/Key** errors prevent it (click “*L.2.6 How do I fix “Key/Index with duplicate values” when saving my Table design?*”).
5. Right after the Table design has been saved (i.e., after point **4**), you should fix **each and every error** that was reported, and that you **carefully noted**. Fixing each error will usually just imply changing the Table data, but on some cases, it may also require to correct/adjust the Table design. The usual way to correct the Table data is by doing a bulk-change of your Table data (click *E.7*).

For each specific cause of data integrity problems, I now detail the specific **error** messages that you may get in step **1** above, together with some advice on how to **fix** the corresponding problem:

- You configured “**Required=Yes**”:  
Click “*L.2.1.1 How do I fix “Nulls in Required field” when saving my Table design?*”.
- You configured a **field validation rule**:  
Click “*L.2.1.2 How do I fix “field validation rule violated” when saving my Table design?*”.
- You configured a **record validation rule**:  
Click “*L.2.1.3 How do I fix “record validation rule violated” when saving my Table design?*”.
- You **changed** the field **type** and/or field **size**:  
Click “*L.2.1.4 How do I fix field Type-size change errors when saving my Table design?*”.

Notice that in this listing there is no bullet for having configured “**Allow Zero Length=No**” because **MS-Access does not check this!** This is bad, and you will have to **verify manually** that there are **no pre-existing zero-length strings**, replacing them in case they do exist: if you want do this, you may click “*E.7 How do I bulk-change my Table/Form’s data?*”.

If you clicked on “**No**” to the question by MS-Access of checking existing data (at the

---

<sup>162</sup> Closing the confirmation window clicking on its “**X**” icon is the same as clicking on “**Cancel**”.

beginning of this section), or you clicked on “**Cancel**” to the question from bullet 2 above, it is very convenient that you **correct** this. You may click:

- “L.2.1.5 How do I fix that I cancelled the Table data integrity check?”

#### L.2.1.1 How do I fix “Nulls in Required field” when saving my Table design?

For **each** Table field configured as “**Required=Yes**” that contains **one or more Null(s)** in the Table, you will get **one** error message saying:

*“Existing data violates the new setting for the 'Required' property for field 'Field\_name.'*

*Do you want to keep testing with the new setting?*

*\* To keep the new setting and continue testing, click Yes.*

*\* To revert to the old setting and continue testing, click No.*

*\* To stop testing, click Cancel.”*

You **fix** this by removing **all Nulls** from field 'Field\_name' indicated in the error message.

Alternatively, you may change the field configuration to “**Required=No**”, but you have to be **very sure** that this is what you want.

If you want to know how to remove all **Nulls** from a field, you may click “E.7 How do I bulk-change my Table/Form's data?”.

#### L.2.1.2 How do I fix “field validation rule violated” when saving my Table design?

For **each** field whose **field validation rule** is violated (i.e., it returns *False*) in the Table, you will get **one** error message saying:

*“Existing data violates the new setting for the 'Validation Rule' property for field 'Field\_name'*

*Do you want to keep testing with the new setting?*

*\* To keep the new setting and continue testing, click Yes.*

*\* To revert to the old setting and continue testing, click No.*

*\* To stop testing, click Cancel.”*

You **fix** this by either correcting the field validation rule you just changed or correcting **all** the field values that violate the field validation rule.

If you want to know how to correct all the wrong values, you may click “E.7 How do I bulk-change my Table/Form's data?”.

#### L.2.1.3 How do I fix “record validation rule violated” when saving my Table design?

If the **record validation rule** is violated (i.e., it returns *False*) by **one or more** Table records, you will get the error message:

**“Existing data violates the new record validation rule.**

*Do you want to keep testing with the new rule?*

*\* To keep the new rule and continue testing, click Yes.*

*\* To revert to the old rule and continue testing, click No.*

*\* To stop testing, click Cancel.”*

You **fix** this by either correcting the record validation rule you just changed or correcting **all** the field values that violate the record validation rule.

If you want to know how to correct all the wrong values, you may click “[E.7 How do I bulk-change my Table/Form’s data?](#)”.

#### **L.2.1.4 How do I fix field Type-size change errors when saving my Table design?**

If saving your Table design, you get either of the following **warning** messages:

- **“Some data may be lost.**  
*The size of one or more fields has been changed to a shorter size. If data is lost, validation rules may be violated as a result. Do you want to continue anyway?”*
- **“Microsoft Access encountered errors when converting the data.**  
*The contents of fields in N record(s) were deleted. Do you want to proceed anyway?”*
- **“Microsoft Access deleted n indexes on the converted fields.**  
*Some data did not convert properly.”*

this is most likely because you **modified** the “**Field Type**” and/or “**Field Size**” properties of one, or more fields of your Table.

This a somehow complex issue: to understand why one of these messages is being shown and the **relevant** consequences of **accepting or not** each of the confirmations above, you may click “[I.4.4.1 What are the side effects of changing the “Field Type” and/or “Field Size” properties of a Table field?](#)”.

#### **L.2.1.5 How do I fix that I cancelled the Table data integrity check?**

In case you cancelled **all or part** of the error checks when saving a Table design (click [L.2.1](#)), you may have data integrity errors in that Table: if your Table data violates the Table design, this will most likely create confusion, misunderstandings and errors in your database data and Query results.

You **fix** this by **re-doing** the Table configuration that was unchecked, **saving** the Table design, and **making sure** that you **do not cancel any error check** (click [L.2.1](#)). For example, if you want to check an index for duplicates, delete the index configuration, save the Table design, open it again, configure the index again and allow MS-Access to check for errors.

### L.2.2 How do I **fix orphan Calculated fields** when **saving my Table design**?

If saving your Table design, you get the **warning** message:

*“There are calculated columns in this table that depend on the column 'Col\_name'. Changing or deleting this column may cause errors in one or more of the dependent calculated columns. Do you want to continue?”*

this is because you deleted a field that is used in the expression of a **Calculated** field.

You **fix** this by either **adding** the missing field, correcting the expression of the **Calculated** field or removing the **Calculated** field, as corresponds.

### L.2.3 How do I **fix “Calculated column cannot be saved...”** when **saving my Table design**?

If saving your Table design, you get the **error** message:

*“Calculated column cannot be saved without a valid expression in the expression property.”*

this is because the expression of a **Calculated** field is wrong. MS-Access will not allow you to save the Table design and will show an additional **warning** message indicating that design changes were not saved.

You **fix** this by correcting the expression of the **Calculated** field or removing the **Calculated** field, as corresponds.

### L.2.4 How do I **fix “Could not find field...”** when **writing the expression of a Calculated field**?

If you write an expression for a **Calculated** field in its property “**Expression**”, and you get the error message:

*“Could not find field 'Field\_name'.”*

this is most likely because you misspelled the field name “*Field\_name*”.

You **fix** this by editing the expression and correcting the typo.

If the field name was **correctly** written, this is most likely because you added the field name “*Field\_name*” to the Table **after** having created the **Calculated** field.

You **fix** this by **saving** the Table and then writing **again** the desired expression for your **Calculated** field. If this does not work, you should try closing and opening the Table, and trying again.

### L.2.5 How do I **fix “Could not find field...”** when **saving my Table design**?

If saving your Table design, you get the error message:

*“Could not find field 'Field\_name'.”*

this is most likely because you removed a field “*Field\_name*” that was used in a **Calculated** field and/or in the Table’s record validation rule.

You **fix** this by doing either of the following:

- **Correct** the corresponding **Calculated** field(s) and/or the Table's record validation rule so they refer to existing field names.
- **Add** a field (creating or renaming it) with the same field name as the missing one.
- **Remove** the corresponding **Calculated** field(s) and/or Table's record validation rule. This is **risky**, so you have to be very sure that this is what you want.

### L.2.6 How do I **fix** "Key/Index with duplicate values" when **saving my Table design**?

If saving your Table design, you get the error message:

*"The changes you requested to the table were not successful because they would create duplicate values in the index, primary key, or relationship. Change the data in the field or fields that contain duplicate data, remove the index, or redefine the index to permit duplicate entries and try again."*

this is most likely because you just configured the **Primary Key** and/or an **index(es) without duplicate values**, and the Table contains records with duplicate values in the **Primary Key field(s)** and/or in the field(s) configured as **indexed without duplicate values**.

You **fix** this by first clicking on "**OK**"<sup>163</sup> to remove the error message. MS-Access will then show the warning message:

*"Errors were encountered during the save operation. Indexes were not added or changed."*

You now click again on "**OK**" to remove the warning message. The Table will remain on "**Design View**". You can now either:

- **Discard** all the design changes by closing the Table and answering "**No**" upon the question on whether you want to save the design changes.
- **Remove** the configuration of the **Primary Key** and/or of the index(es) that caused the error.

After having done either of the above, you can open the Table in "**Datasheet View**" and modify (or remove) the records with duplicate values that originated the problem. If you want to know how to correct the erroneous records, you may click "[E.7 How do I bulk-change my Table/Form's data?](#)". Once the records are correct, you may configure again the **Primary Key** and/or the index(es) **without duplicate values** that you wanted to configure.

Notice that the error message above is exactly the same as when entering a record with duplicate values in the fields of an index **without duplicate values** or in the **Primary Key fields** (click L.4.3.2).

---

<sup>163</sup> Clicking on the close window "**X**" icon of the error window will have the same effect as clicking on "**OK**".



### L.2.7 How do I fix “...primary key cannot contain a Null...” when saving my Table design?

If saving your Table design, you get the error message:

*“Index or primary key cannot contain a Null value.”*

this is most likely because you just configured the **Primary Key**, and the Table contains records with **Null** in one or more **Primary Key field(s)**.

You **fix** this by first clicking on “**OK**”<sup>164</sup> to remove the error message. MS-Access will then show the warning message:

*“Errors were encountered during the save operation. Indexes were not added or changed.”*

You now click again on “**OK**” to remove the warning message. The Table will remain on “**Design View**”.

MS-Access **will not allow** you to open the Table in “**Datasheet View**” (to modify the records) while the conflicting **Primary Key** is set. You therefore have **first** to **remove** the configuration of the **Primary Key**. After having done this, you can then open the Table in “**Datasheet View**” and modify (or remove) the records with **Null** that originated the problem. If you want to know how to correct the erroneous records, you may click “*E.7 How do I bulk-change my Table/Form’s data?*”. Once the records are correct, you may configure again the **Primary Key** that you wanted.

### L.2.8 How do I fix “You can’t change the primary key” when saving my Table design?

If saving your Table design, you get the error message:

*“You can’t change the primary key.*

*This table is the primary table in one or more relationships*

*If you want to change or remove the primary key, first delete the relationship in the relationship’s window.”*

this is most likely because you tried to remove/change the **Primary Key field(s)** which constitute(s) the **index without duplicates and without Nulls** associated with the **master field(s)** of a Relationship.

If you really want to remove/change the **Primary Key field(s)**, you **fix** this by first removing the corresponding Relationship(s). You will then be able to remove/change the **Primary Key field(s)**.

---

<sup>164</sup> Clicking on the close window “**X**” icon of the error window will have the same effect as clicking on “**OK**”.

### L.2.9 How do I **fix** “**Cannot delete this index...**” when **saving** my **Table design**?

If saving your Table design, you get the error message:

*“Cannot delete this index or table. It is either the current index or is used in a relationship.”*

this is most likely because you tried to remove/change the **index without duplicates and without Nulls** associated with the **master** field(s) of a Relationship.

If you really want to remove/change this **index without duplicates and without Nulls Primary Key field(s)**, you **fix** this by first removing the corresponding Relationship(s). You will then be able to remove/change the index.

### L.2.10 How do I **fix** “**There is no primary key...**” when **saving** my **Table design**?

If you have **created a new** Table, and when **saving the Table design** you get the **warning** message:

***“There is no primary key defined***

*Although a primary key isn't required, it's highly recommended. A table must have a primary key for you to define a relationship between this table and other tables in the database.*

*Do you want to create the primary key now?*

this is because you did not define the **Primary Key** in your Table design. It is not an error to create a Table without defining its **Primary Key**, but it is **very unusual**. Unless you are **very sure** you **do not** want a **Primary Key** in this Table, my advice is that you configure the field(s) that define it. Depending on what you want to do, you may take one of the following options:

- If you **do not** want a **Primary Key** in this Table, click on “**No**” and the Table design will be saved without having a **Primary Key**. Notice that **you will not** get a warning the subsequent times that you save the Table design: you **only** the warning above once, which is **the first time** that you save the Table design.
- If you **want** to define now the **Primary Key field(s)** of this Table, click on “**No**” to save the Table design changes. You then open the Table in “**Design View**” and define the **Primary Key field(s)** of the Table. If you want to know how to do it, you may click “*D.6 How do I configure the Primary Key field(s) of a Table?*”.
- If you click on “**Yes**”, MS-Access will add to your Table an **AutoNumber** field, will configure it as the **Key field**, and will leave the Table open in “**Design View**”. My advice is you **do not take this option**, because it is much better to define the **Primary Key field(s)** based on the intrinsic nature of the Table fields and avoid using the **AutoNumber** fields. If you want to know more about this, you may click “*D.6 How do I configure the Primary Key field(s) of a Table?*”.

## L.3 How do I fix errors in my Relationship configuration?

You may click:

- “L.3.1 How do I fix “different number of fields” Relationship error?”
- “L.3.3 How do I fix “different field sizes” Relationship error?”
- “L.3.4 How do I fix “violates referential integrity” Relationship error?”
- “L.3.5 How do I fix “missing index” Relationship error?”
- “L.3.6 How do I fix “...could not lock table...” Relationship error?”
- “L.3.7 How do I fix a removed Relationship when I create a new one?”
- “L.3.8 How do I fix showing a wrong Relationship type?”
- “L.3.9 How do I fix “reversed Relationship Tables and fields” error?”
- “L.3.10 How do I fix that I cannot configure a Relationship between a Table and itself?”
- “L.3.11 How do I fix a malfunctioning Table slave record?”
- “L.3.12 How do I fix a malfunctioning Relationship?”

### L.3.1 How do I fix “different number of fields” Relationship error?

If clicking on “**OK**” or “**Create**” in an “**Edit Relationships**” box, you get the error message:

*“The field name is missing in row n.*

*You haven't selected a matching field for this relationship in each row of the grid.*

*Select fields so that the grid has the same number of fields on the left and right sides, and then try to create the relationship again.”*

this is most likely because the **number** of **master** fields and the **number** of **slave** fields is not the same. The error message indicates explicitly the row number (indicated as “n” in the error message above) within the Relationship box where the missing field name is located.

You **fix** this by first clicking on “**OK**” to remove the error message.

Then, you modify the fields in the “**Edit Relationships**” box so that each master field is related to a slave field (and vice versa), and both fields in each master-slave pair have the same field type and field size.

Remind also that if the Relationship is with referential integrity, the master field(s) should have one joint index **without duplicate values** and **without Nulls**. Once you are done, click on “**OK**” or “**Create**” to save the Relationship.

Alternatively, you may cancel the operation by clicking on “**Cancel**” in the “**Edit Relationships**” box: if you were creating a new Relationship, it will not be created; if you were editing an existing Relationship, the existing Relationship will remain unaltered.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5* and *B.10.7*.

### L.3.2 How do I fix “different field types” Relationship error?

If clicking on “OK” or “Create” in an “Edit Relationships” box, you get the error message:

*“Relationship must be on the same number of fields with the same data types.”*

this is most likely because one, or more, of the master-slave field pair(s) involved in the Relationship have a **different** field type.

You **fix** this by first clicking on “OK” to remove the error message.

Then, you modify the fields in the “Edit Relationships” box so that both fields in each master-slave pair have the same field type and field size.

Remind also that if the Relationship is with referential integrity, the master field(s) should have one joint index **without duplicate values** and **without Nulls**. Once you are done, click on “OK” or “Create” to save the Relationship.

Alternatively, you may cancel the operation by clicking on “Cancel” in the “Edit Relationships” box: if you were creating a new Relationship, it will not be created; if you were editing an existing Relationship, the existing Relationship will remain unaltered.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5* and *B.10.7*.

### L.3.3 How do I fix “different field sizes” Relationship error?

If clicking “OK” or “Create” in an “Edit Relationships” box (with a ticked “Enforce Referential Integrity” checkbox), you get the following **wrong** error message:

*“Microsoft Access can't create this relationship and enforce referential integrity.*

*\* The fields you chose may have different data types.*

*\* The fields may have the Number data type but not the same FieldSize property setting.*

*Try one of the following:*

*\* Select fields with the same data type.*

*\* Open the tables in Design view, and change the data types and field sizes to that the fields match.*

*If you want to create the relationship without following the rules of referential integrity, clear the Enforce Referential Integrity check box.”*

this is most likely because of either (or both):

- One (or more) of the master-slave field pair(s) involved in the Relationship have the same field type, but a **different field size**.
- You mistakenly typed-in **the same** master-slave field pair **twice** in the “Edit Relationships” box.

You **fix** this by editing your Relationship to make all master-slave field pairs having the same field type and field size and/or removing one of the two duplicated **master-slave** field pairs.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5*

and B.10.7.

### L.3.4 How do I fix “violates referential integrity” Relationship error?

If clicking on “OK” or “Create” in an “Edit Relationships” box (with a ticked “Enforce Referential Integrity” checkbox), you get the error message:

**“Microsoft Access can't create this relationship and enforce referential integrity.**

*Data in the table 'Table\_name' violates referential integrity rules.*

*For example, there may be records relating to an employee in the related table, but no record for the employee in the primary table.*

*Edit the data so that records in the primary table exist for all related records.*

*If you want to create the relationship without following the rules of referential integrity, clear the Enforce Referential Integrity check box.”*

this is most likely because there are **records** in the **slave** Table 'Table\_name' whose **slave field(s)** contain values that **are not equal** to the ones of the corresponding **master field(s)** in **any** record in the master Table.

You **fix** this by first clicking on “OK” to remove the error message.

If the Relationship definition was wrong, you correct it. If the Relationship was correct, then you have to **fix** the records in the **master** Table and/or in the **slave** Table. If you want to know how to correct the erroneous records, you may click “E.7 How do I bulk-change my Table/Form's data?”. Once the records are correct, you may configure again the Relationship that you wanted.

Alternatively, you may cancel the operation by clicking on “Cancel”: if you were creating a new Relationship, it will not be created; if you were editing an existing Relationship, the existing Relationship will remain unaltered.

If you want to know how to **edit** and **delete** Relationships, you may click B.10.5 and B.10.7.

### L.3.5 How do I fix “missing index” Relationship error?

If clicking on “OK” or “Create” in an “Edit Relationships” box (with a ticked “Enforce Referential Integrity” checkbox), you get the error message:

**“No unique index found for the referenced field of the primary table.”**

this is most likely because the **master** field(s) do not have one (composite) index **without duplicate values and without Nulls**.

You **fix** this by first clicking on “OK” to remove the error message.

You then have to configure one index without duplicate values and without **Nulls** over the master field(s). This implies that all master fields should be configured as “Required=Yes”, they should contain no **Nulls**, and you should also configure one index without duplicate values over the **master** fields. Remind that if the **master** fields are a **Primary Key**, this always implies that they have one index **without duplicate values and without Nulls**. Remind also that you may configure **several** indexes in a Table, in addition to its **Primary Key field(s)**. If you want more information on this, you may click

*“D.7.2 How do I add composite (and simple) indexes to a Table?”*

Alternatively, you may cancel the operation by clicking on **“Cancel”**: if you were creating a new Relationship, it will not be created; if you were editing an existing Relationship, the existing Relationship will remain unaltered.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5* and *B.10.7*.

### **L.3.6 How do I fix “...could not lock table...” Relationship error?**

If clicking on **“OK”** or **“Create”** in an **“Edit Relationships”** box, you get the error message:

*“The database engine could not lock table 'Table\_name' because it is already in use by another person or process.”*

this is most likely because the table *“Table\_name”* is opened by some database user.

You fix this by first clicking (s) on the **“OK”** button or the close **“X”** icon to remove the error message. You then close the Table *“Table\_name”* or contact the user that is using it, so he/she closes it. You can now **retry** to create the Relationship by clicking on **“OK”** or **“Create”** in the **“Edit Relationships”** box.

### **L.3.7 How do I fix a removed Relationship when I create a new one?**

If you **create a new** Relationship and at that moment an existing Relationship disappears, this is most likely because you were **editing** the Relationship that has disappeared instead of creating a new one. This error is particularly frequent when you open a **blank “Edit Relationships”** box by double-clicking on the background of the **“Relationships”** pane and you think you have created a new Relationship, but this is **not** true. You have opened a **blank “Edit Relationships”** box where you will be able to **select one existing Relationship** to edit it. The way to know if you are editing an existing Relationship or creating a new one is to check the top right button of the **“Edit Relationships”** box: if the button is labeled **“OK”**, you are **editing** an **existing** Relationship; if it is labeled **“Create”**, you are **creating** a **new** Relationship.

To start creating a **new** Relationship you have to either drag and drop a **master** field into its **slave** field, or click on the **“Create New..”** button from any **“Edit Relationships”** box.

If you want to **create** a new Relationship, you may click *“D.9.1 How do I create a new Relationship?”*.

If you want to know how to **manage** Relationships, you may click *“B.10 What is the “Relationships” pane?”*.

### **L.3.8 How do I fix showing a wrong Relationship type?**

If MS-Access is **showing** a **one-to-one** Relationship as a **one-to-many** Relationship (i.e., shown with and **infinity** symbol **“∞”** in the **line endpoint** of each **slave** field), this is most likely because you configured the required index **without duplicate values and without Null** in a **subset** of the **slave field(s)** **after** having created the Relationship. In this, case, MS-Access will wrongly continue showing the Relationship as when it was created: a **one-to-many** Relationship pane. This may be an MS-Access **bug**.

You fix **this** by **deleting** the Relationship (click *B.10.7*) and **adding** it again (click *D.9.1*). I have tried many alternatives to this (refreshing, closing and opening the file, clicking on “**Compact and Repair Database**”, etc.) and nothing worked.

### L.3.9 How do I fix “reversed Relationship Tables and fields” error?

This also answers the question:

- “**Why is MS-Access reversing my master and slave Table-boxes?**”

If you create a new Relationship (either using the “**Create New..**” button or by drag-and-drop of one **master** field), it may happen that in the resulting “**Edit Relationships**” box MS-Access has **reversed** the first **master** field and the first **slave** fields<sup>165</sup> that you entered to start configuring the Relationship, and MS-Access **will not** allow you to change them back.

When I say that MS-Access has **reversed** them it means that the **slave** field and the **slave** Table are on the **left** column of the “**Edit Relationships**” box, while the **master** field and the **master** Table are on the **right** column. Remind that in the “**Edit Relationship**” box, the **master** Table and fields are the ones on the **left** side of the box, and the **slave** Table and **fields** are the ones on the right side of the box.

This happens when the **slave field is configured** with a **simple index without duplicate values** (with or without **Nulls**) and the **master field is not configured** with a **simple index without duplicate values and without Nulls**.

The reason for **reversing** the Table-boxes (and also the fields) is that MS-Access **guesses** that you are trying to create a **one-to-many** Relationship, and therefore, the field that is a **candidate Key must** be the **master** field. This is not a problem, but it can be **very puzzling** if you are not aware of it.

The guess of MS-Access is right, **except** when you are trying to **create a multi-field one-to-one** Relationship. If this is the case, **both the master fields and the slave fields** are a **composite candidate Key** of their respective Tables. Therefore, when you **finished** inputting **all** the master and slave fields, it **would have been correct** to use the **master** Table-box and the **first** master field that you used.

If MS-Access’ guess was **right**, and you made a **mistake** with the **first** master Table-box and master field, the decision by MS-Access of reversing the Table-boxes and fields is great, because it is **correcting** your mistake.

If MS-Access’ guess was **wrong**, and you were trying to create a multi-field **one-to-one** Relationship, you **fix** this by selecting as **the first slave** field that you use to **start** creating the Relationship, **one** of the slave fields of the Relationship **that is not indexed without duplicate values**. Doing this, MS-Access will **create** the “**Relationship box**” with the **master** and **slave** Tables and fields that you want, and **afterwards** you will be able to add the remaining **master** fields and **slave** fields that you wanted for the Relationship, even if one or more of the **slave** fields has a **simple index without duplicate values** (with or without **Nulls**).

In case that **all** the **slave** fields **have a simple index without duplicate values** (with or

---

<sup>165</sup> Notice that if the **master** and **slave** fields have been reversed, this implies the master and slave Tables have also been reversed.

without **Nulls**), in order to apply the **fix** I have just indicated, you have to **first remove the simple index** of **one** of the **slave** fields, then apply the fix I just indicated above, and afterwards configure again the **simple** index of the corresponding **slave** field.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5* and *B.10.7*.

### **L.3.10 How do I fix that I cannot configure a Relationship between a Table and itself?**

You cannot do it because you are creating the Relationship by doing drag-and-drop of the **master** field to the **slave** field **in the same Table-box**.

You **fix** this using either of the two following options:

- You create the Relationship clicking on the “**Create New..**” button in any “**Edit Relationship**” box. When the “**Create New**” box is shown, you can select the Table that you want as both the **master** Table and the **slave** Table, even if both are the same Table.
- You show a **second** Table-box for the Table (e.g., by doing drag-and-drop from the Table name in the “**Navigation Pane**” to the “**Relationships**” pane, and then, doing drag-and-drop of the **master** field name **from one** of the Table-boxes to the **slave** field **in the other** Table-box.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5* and *B.10.7*.

### **L.3.11 How do I fix a malfunctioning Table slave record?**

If a slave record in a Relationship with referential integrity is taking wrong values, click “*L.3.12 How do I fix a malfunctioning Relationship?*”.

### **L.3.12 How do I fix a malfunctioning Relationship?**

If the values of the **slave fields** in one (or more) **slave records** do not match the ones of the corresponding **master fields** in any of the records from its master Table, in a Relationship with referential integrity, this is most likely because the **slave fields** are involved as **slave** fields in **more than one** Relationship with referential integrity. Therefore, the **slave fields** may take the values corresponding to **master records** from **any of its master Tables**. If you are looking at **one** of its **master Tables**, it can be the case that the **slave record** is not taking the values from any of the records of the **master** Table you are looking at, and rather it is taking them from **another** of its master Tables.

It is **most unusual** that you actually want to have a slave field with **more than one** master field. My advice is you suppress the additional Relationships and each **slave field** has **one and only one master field**.

You **fix** this by **removing** all the additional Relationships with referential integrity, such that **each slave field** is involved in **one and only one** Relationship with referential integrity.

If you want to know how to **edit** and **delete** Relationships, you may click *B.10.5* and *B.10.7*.



## **L.4 How do I fix errors when entering, modifying or deleting records?**

You may click:

- “*L.4.1 How do I fix various field value errors?*”
- “*L.4.2 How do I fix error messages when saving a field value?*”
- “*L.4.3 How do I fix error messages when entering a record?*”
- “*L.4.4 How do I fix errors when pasting records into a Table/Form?*”
- “*L.4.5 How do I fix a record I cannot delete?*”

### **L.4.1 How do I fix various field value errors?**

You may click:

- “*L.4.1.1 How do I fix that I cannot type-in a text string in full?*”
- “*L.4.1.2 How do I fix that I cannot paste into a field’s value?*”
- “*L.4.1.3 How do I fix a changed numeric value?*”
- “*L.4.1.4 How do I fix a changed Short Text value that I typed-in?*”
- “*L.4.1.5 How do I fix a Table Short Text field configured as “Required=Yes” that accepts a Null?*”
- “*L.4.1.6 How do I fix typing-in invisible characters into a Short Text Table field?*”

#### **L.4.1.1 How do I fix that I cannot type-in a text string in full?**

If you are **typing-in** a text string in a **Short Text** field, and MS-Access **stops** showing the characters that you type, this is most likely because you have reached the maximum field size. MS-Access will not allow you to enter a text string longer than the “**Field Size**” property of the field.

You **fix** this by editing the text string that you are entering to make it shorter or equal than the value of the “**Field Size**” property of the field.

Alternatively, you may cancel the operation by pressing the “**Esc**” key: if you were entering a new record, the new record will be removed; if you were editing an existing record, the existing record will remain unaltered.

Finally, you may want to increase the value of the “**Field Size**” property, and you will then be able to enter in this field text strings up to the length of the newly configured value of “**Field Size**”. This is not risky (unless you have very severe space restrictions), and it is actually considered a **good practice** to always set the “**Field Size**” of **Short Text** fields to the maximum allowed value of 255.

#### **L.4.1.2 How do I fix that I cannot paste into a field’s value?**

If **pasting** one text string into a **field’s value** (click *E.5.2.1*) in a **Short Text** field, you get

the error message:

- “*The text is too long to be edited.*”

this is most likely because the **string** you are trying to paste into the **field's value** would result in a total length of the string **larger** than the “**Field Size**” property of this **Short Text** field. MS-Access does not allow you to edit into a field a text string longer than the “**Field Size**” property of the field.

You **fix** this by pasting a text string that results in a total size which is at most the value of “**Field Size**”.

Finally, you may want to increase the value of the “**Field Size**” property, and you will then be able to enter in this field text strings up to the length of the newly configured value of “**Field Size**”. This is not risky (unless you have very severe space restrictions), and it is actually considered a **good practice** to always set the “**Field Size**” of **Short Text** fields to the maximum allowed value of 255.

#### L.4.1.3 How do I **fix a changed numeric value**?

This section also answers the questions:

- **Why is MS-Access making pasting errors in a numeric value?**
- **Why is MS-Access changing a numeric value that I type-in?**

If **saving** a **numeric** field value, MS-Access shows a **different** value, this is most likely because of either of the following causes:

- The **formatting** configured for the field is **not showing** the actual **saved** field value. For example, if you configure the field format to show only “**n**” decimal digits, and you saved a number with more than “**n**” decimal digits, the number **shown** will be rounded (using round-half away from zero, click *J.11.20*) to “**n**” decimal digits.

This is not an error as such, but if you want to **fix** this, change the formatting of the field to the one that you find more suitable. You may click “*H.6 How do I configure the formatting of column values in a Table/Query/Form?*”.

- The value you tried to **save** is fractional (e.g., 45.56) and the field type-size only admits integer values (e.g., the field type-size is **integer** or **double**). Therefore, the **saved** value is the one you tried to save **rounded** with round-half away from zero (click *J.11.20*).
- The value you tried to **save cannot** be represented with all its precision in the field **type** and **size**. Therefore, the **saved** value is the one you tried to save **approximated** to the **nearest** value that can be represented in the field type and size. If you want to know more about field storage formats, you may click “*G.9.1 How are numeric-like values internally represented?*” and “*G.9.3 What are decimal/binary conversion rounding errors?*”. I am adding some examples below, to clarify a few frequent cases.

You **fix** this by changing the field type-size to another one, but you have to be **very sure** because this is **very risky** (click *I.4.4.1*).

Some examples of this problem are:

- For a field with an **integer field type** (“**Field Type=Number**” and “**Field Size**” set

to **Byte, Integer, Long Integer, Decimal** or **Large Number**), if you try to **save** a **fractional** number, then the **saved** value is the fractional number **rounded** using half-round to even (click *J.11.20*).

- For a field with the **Currency field type**, if you try to **save** a number with more than four decimal positions, then the **saved** value is the number you tried to save rounded to four decimal positions (using half-round to even, click *J.11.20*).
- For a field with a **floating-point field type** (“**Field Type=Number**” and “**Field Size**” set to **Single** or **Double**), if you **enter** a number with more precision (i.e., more significant digits) than the one that can be stored, MS-Access will change the value and store/calculate what is possible with the available precision. For example, if you type in **.9** into a **Number-Single** field, it will become **.899999976158142**. Likewise, if you type in **.1** into a **Number-Single** field, it will become **.100000001490116**.

#### **L.4.1.4 How do I fix a changed Short Text value that I typed-in?**

If you **type-in** a text-string in a **Short Text** field, and MS-Access **changed** its value, this is most likely because MS-Access **removes any trailing space characters** that you typed-in. This is a good feature, because invisible characters are a source of errors. In case you want more detail about problems with text strings, you may click “*L.7 How do I fix errors with Short Text or String fields?*”.

It can also be that the field width and height is not enough to **show** the whole text string, and only part of it is being shown.

#### **L.4.1.5 How do I fix a Table Short Text field configured as “Required=Yes” that accepts a Null?**

If you have a **Short Text** field configured as “**Required=Yes**” and you can **edit Null** into it, this is most likely because you are **not** editing (e.g., pasting) a **Null**, and rather, you are editing either the **zero-length** text string (click *L.7.7*) or an **invisible** text string (click *L.7.8*).

There is nothing to fix in this case. If you want to check that everything is right, enter an actual **Null** and you will see how MS-Access rejects it.

#### **L.4.1.6 How do I fix typing-in invisible characters into a Short Text Table field?**

I clearly advise you **avoid typing-in any invisible character** (except initial or intermediate spaces) in your **Short Text** Table fields.

If you **do not type-in anything** into a Table **Short Text** field, it will contain **Null** (unless you configured a default value for this field).

If against my advice you want to type-in invisible characters, you may click “*L.7.6 What are invisible characters?*”.

If against my advice you want to type-in the zero-length string, you may click “*L.7.7 What is the zero-length text string?*”.

## L.4.2 How do I fix error messages when saving a field value?

This section addresses error messages that you may get when **saving a field value** (either typed-in, chosen, or pasted) within a record. This may happen when interactively editing records (click *E.1*), when pasting over a rectangle of fields over more than one record (click *E.5.2.2*) or when pasting more than one row of values as new records (click *E.5.2.3*).

You may click:

- “L.4.2.1 How do I fix an invalid value?”
- “L.4.2.2 How do I fix trying to save Null in a “Required” field?”
- “L.4.2.3 How do I fix violating “Allow Zero Length=No”?”
- “L.4.2.4 How do I fix that I cannot paste a text string in full?”
- “L.4.2.5 How do I fix a value rejected because it is not in the list?”
- “L.4.2.6 How do I fix a value violating a field validation rule?”

### L.4.2.1 How do I fix an invalid value?”

If **saving a field value**, you get the **error** message:

***“The value you entered isn’t valid for this field.***

*For example, you may have entered text in a numeric field or a number that is larger than the FieldSize setting permits.”*

this is most likely because you tried to **save** a value that **cannot** be represented in the field type and size of the field:

- The **value** does **not match** the **field type** and **size**:  
You cannot enter an arbitrary text string (e.g., “**Chicago**”) into a **Number**, **Currency** or **Large Number** fields. You cannot enter a **Date/Time** value (e.g., **23/12/2018**) into a **Number**, **Currency** or **Large Number** fields. You **cannot** enter a numeric value into a **Date/Time** field<sup>166</sup>. As an exception, you can enter any number into a **Yes/No** field: “**0**” is stored as “**0**” (i.e., **False/No/Off** or **unticked**) and any other number you enter is stored as “**-1**” (i.e., **True/Yes/On** or **ticked**).
- The value is **out of the range** of the **field type**:  
A **Number-Byte** value must be between **0** and **255** (both included). A **Number-Integer** value must be between **-32,768** and **32,767** (both included). If you want to know the specific ranges of every field-type, you may click “*G.2 How do I manage VBA data types and Table field types-sizes?*”.
- A **Date/Time** value that does not exist:  
For example, “**30-february-2018**” or “**25:67**” do not exist.
- A **Short Text** value longer than “**Field Size**”  
The string length is larger than the value of the “**Field Size**” property of this field.

---

<sup>166</sup> Since MS-Access accepts “,” as date separator and “.” as time separator, you can enter **some values that look like numbers, but they are not**, into a **Date/Time** field.

You **fix** this by **editing** another field value that is within the range of the corresponding field type-size.

Finally, you may want to modify the type-size of this field in the Table design, in order to accept the value that you want to enter, but you have to be **very sure** because this is **very risky** (click *I.4.4.1*).

#### **L.4.2.2 How do I fix trying to save Null in a “Required” field?**

If **saving a field value**, you get the **error** message:

*“You must enter a value in the 'Table\_name.Field\_name' field.”*

this is most likely because you tried to **save Null** in a field configured as “**Required=Yes**”. The corresponding field name is “*Field\_name*” from the Table “*Table\_name*”, both of them explicitly include in the error message.

You **fix** this **editing** a **non-Null** value into the field.

Finally, you may want to change the field configuration setting it to “**Required=No**”, but you have to be **very sure** because this is **very risky** (click *I.4.4.2*).

#### **L.4.2.3 How do I fix violating “Allow Zero Length=No”?**

If **saving a field value**, you get the **error** message:

*“Field 'Table\_name.Field\_name' cannot be a zero-length string.”*

this is most likely because you tried to **save** a zero-length string in a **field** configured as “**Allow Zero Length=No**”. The corresponding field name is “*Field\_name*” from the Table “*Table\_name*”, both of them explicitly include in the error message.

You **fix** this by **editing** a non-zero string in the field.

Finally, you may want to modify the field configuration setting it to “**Allow Zero Length=Yes**”, but you have to be **very sure** because this is **very risky** (click *I.4.4.2*).

#### **L.4.2.4 How do I fix that I cannot paste a text string in full?**

If **pasting** one text string over one **Short Text** field (click *E.5.2.2*), you get following **error** message:

- *“The field is too small to accept the amount of data you attempted to add. Try inserting or pasting less data.”*

this is most likely because the **string** you are trying to **paste** into the field is **longer** than the “**Field Size**” property of this **Short Text** field. MS-Access does not allow you to edit a text string longer than the “**Field Size**” property of the field.

You **fix** this by pasting a text **string** shorter or equal than the value of “**Field Size**”.

Finally, you may want to increase the value of the “**Field Size**” property, and you will then be able to enter in this field text strings up to the length of the newly configured value of “**Field Size**”. This is not risky (unless you have very severe storage restrictions): it is actually considered a **good practice** to set “**Field Size**” to the maximum allowed value of 255.

#### L.4.2.5 How do I fix a value rejected because it is not in the list?

If **saving a field value**, you get the **error** message:

*“The text you entered isn't an item in the list.*

*Select an item from the list, or enter text that matches one of the listed items.”*

this is most likely because you tried to **save** a value that is **not** one of the options of the field’s drop-menu, and the drop-down menu is configured as “**Limit to List=Yes**”.

You **fix** this by **editing** one of the options from the drop-down menu.

Finally, you may want to configure the property “**Limit to List=No**”, but you have to be **very sure** because this **may be risky**. If you want to know more about this, you may click “*D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field?”*”.

#### L.4.2.6 How do I fix a value violating a field validation rule?

If **saving a field value**, you get the **error** message:

*“One or more values are prohibited by the validation rule 'Boolean\_expr'. Enter a value that the expression for this field can accept.”*

this is most likely because you tried to **save** a value that violates (i.e., it returns **False**) the field validation rule. The **expression** of the field validation rule is explicitly listed in the error message, represented as “*Boolean\_expr*” above.

In case that you configured a **specific error message** in the **field** property “**Validation Text**” (click *D.5.1.6*), upon an error you will get this **specific error message instead** of the **default** error message listed above.

You **fix** this by changing the field value you just entered to other one that respects the field validation rule.

Finally, you may want to modify the field validation rule, but you have to be **very sure** that changing the rule is the correct solution.

#### L.4.3 How do fix error messages when entering a record?

This section addresses error messages that you may get when **entering** or **modifying** a record (click *Part E*)

You may click:

- “*L.4.3.1 How do I fix a record violating a record validation rule?”*”
- “*L.4.3.2 How do I fix erroneous records because of duplicate values?”*”
- “*L.4.3.3 How do I fix a slave record without a master record?”*”
- “*L.4.3.4 How do I fix an erroneous slave record?”*”

##### L.4.3.1 How do I fix a record violating a record validation rule?

If **entering** or **modifying a record**, you get the **error** message:

*“One or more values are prohibited by the validation rule 'Boolean\_expr' set for*

*'Table\_name'. Enter a value that the expression for this field can accept.'*

this is most likely because the record you intended to **enter**, or your intended **modified** record, violates the Table's record validation rule (i.e., it returns *False*). The expression of the field validation rule ("*Boolean\_expr*" above) and the Table name ("*Table\_name*" above) are explicitly listed in the error message.

In case that you configured a **specific error message** in the **record** property "**Validation Text**" (click *D.8.2*), upon an error you will get this **specific error message** instead of the **default** error message listed above.

You **fix** this by changing the field value(s) of the intended record to other value(s) that satisfy the record validation rule.

Finally, you may want to modify the record validation rule, but you have to be **very sure** that changing the rule is the correct solution (click *I.5.7*).

#### **L.4.3.2 How do I fix erroneous records because of duplicate values?**

If **entering** or **modifying** a **record**, you get the **error** message:

*"The changes you requested to the table were not successful because they would create duplicate values in the index, primary key, or relationship. Change the data in the field or fields that contain duplicate data, remove the index, or redefine the index to permit duplicate entries and try again."*

this is most likely because the record you intended to **enter**, or your intended **modified** record, has **the same value(s)** as the ones of an **already existing record** in the field(s) of the **Primary** key, or of an **index without duplicate values**.

You **fix** this by changing the field value(s) of the intended record to other one(s) that is(are) **not duplicate** in respect to the one(s) of any other record already existing in the Table. Alternatively, you may cancel the operation by pressing the "**Esc**" key: if you were entering a new record, the new record will be removed; if you were editing an existing record, the existing record will remain unaltered.

You may also want to edit/remove the already existing record that had duplicate values in respect to the new one you were trying to enter, and then re-enter the record you were trying to enter.

Alternatively, you may want to modify the **Primary Key** or the **index** that was preventing the insertion of the new record, but you have to be **very sure** because this is **very risky** (click *I.5.6*).

Notice that the error message above is exactly the same as when configuring the **Key field(s)** or an **index without duplicate values** in a Table that has duplicate values in those field(s) (click *L.2.6*).

#### **L.4.3.3 How do I fix a slave record without a master record?**

If **entering** or **modifying** a **record**, you get the **error** message:

*"You cannot add or change a record because a related record is required in table 'Table\_name'."*

this is most likely because the record you intended to **enter**, or your intended **modified**

record, has one (or more) **slave** field(s) that have a **value-array** such that there is no **master** record in the **master** Table that has **the same value-array**. The name of the **master** Table is explicitly indicated in the error message, represented as “*Table\_name*” above.

You **fix** this by changing the **slave** field value(s) to other one(s) that exist in a **master** record in the **master** Table.

Finally, you may want to modify the Relationship, but you have to be **very sure** because this is **very risky**.

If you want to know more about this, you may click “*C.11 What is a Relationship?*” and “*D.9 How do I create and configure my Table Relationships?*”.

#### **L.4.3.4 How do I fix an erroneous slave record?**

If **entering** or **modifying** a record, you get the error message:

*“The record cannot be deleted or changed because table 'Table\_name' includes related records.”*

this is most likely because the record you intended to **enter**, or your intended **modified** record, is a **master** record in one (or more) Relationship(s) **with referential integrity** (click “*C.11 What is a Relationship?*”) with the option “**Cascade Update Related Fields**” (click *D.9.4*), and one (or more) of its **slave** record(s) does not pass **all** the error checks (click *E.6*) if updating the values of its **slave** field(s) to the one(s) of the **intended master** record.

You **fix** this by changing the **master** field value(s) to other one(s) that if updated in **all** its **slave** record(s) **all** of them will pass **all** the error checks.

Finally, you may want to modify the Relationship, but you have to be **very sure** because this is **very risky**.

If you want to know more about this, you may click “*C.11 What is a Relationship?*” and “*D.9 How do I create and configure my Table Relationships?*”.

#### **L.4.4 How do I fix errors when pasting records into a Table/Form?**

If your problem is when **pasting one row of values as new records** or when pasting **over a rectangle of fields over one record**, you may click *E.1*. Otherwise, continue reading here.

If you **paste over a rectangle of fields over more than one existing record** (click *E.5.2.2*), or if you **paste more than one row of values** in the **new-record row** (click *E.5.2.3*), MS-Access will **attempt** to **modify** or **enter** (respectively) the corresponding records **one by one**.

For **each record** that is to be **modified** or **entered** MS-Access will **first** perform, a **field error check, field by field**, on **each** field whose value has been **pasted** in the record. The **field error checks performed** are the same ones as when attempting to **save** a field value (click *E.6.1*).

**After** having checked **all** the **pasted** fields of a given record, that **record** as a whole is also checked for errors. The **record error checks** performed are the same ones as when attempting to **enter** or **modify** a record (click *E.6.2*).



If **all** the above **field** and **record checks are correct**, the record is modified or entered. If **all** the **records** are correct, all of them are **modified** or **entered** in the Table/Form.

If one of the above **field/record checks results in an error**, MS-Access will show an **error message** informing you of the **specific** cause of the error (validation rule, type mismatch, etc.) with an “**OK**” button.

Once you click on “**OK**” to close the error message box, if any additional error is encountered, MS-Access will show you the warning message:

***“Do you want to suppress further error messages telling you why records can't be pasted?”***

*If you click No, a message will appear for every record that can't be pasted.”*

inside a dialog box. The options to click are “**Yes**”, “**No**” and “**Cancel**”:

- **If you click “Cancel”** (or you close the confirmation box)  
The **paste** operation is **canceled**, and the Table/Form will remain unchanged.
- **If you click “No”**  
MS-Access will show you **one error box for each record** whose data are **not totally correct**, and in **each and every** error box you will have to click “**OK**” to close it. There is **no way to cancel this**, and you will have to click “**OK**” in **all the error notification boxes**. If the pasted record-list is **long**, this can be really **annoying**. Make sure that you **really** want to see **all** the error messages before clicking “**No**”.

**After** you have clicked “**OK**” **on each and every** error box, MS-Access will continue as if you had clicked “**Yes**”: see the next bullet point.

- **If you click “Yes”**  
If there are **no correct records**, no record will be **entered/modified**.

If there are “**n**” correct records, MS-Access will show the warning message:

*“You are about to paste **n** record(s). Are you sure you want to paste these records?”*

inside a confirmation box. If you click “**Yes**”, the “**n**” correct records will be entered/modified. Notice that MS-Access may **silently change** the values that you have pasted because of storage format restrictions of the field type. If you want to know more about this, you may click [L.4.1.3](#), [L.4.4.1](#), [L.4.4.2](#) or [L.4.4.3](#).

If you click “**No**” no record will be **entered/modified**.

Regardless of having **entered/modified** any correct record or not, in case there was one (or more) erroneous records, MS-Access will finally show the following error message:

***“Records that Microsoft Access was unable to paste have been inserted into a new table called 'Paste Errors'.***

*In the Navigation Pane, open the new table to see the unpasted records.*

*After you fix the problems that resulted in the paste errors, copy and paste the records from the new table.”*

indicating that **all the erroneous** records (i.e., all the records whose data did not pass all

the record error checks) have been pasted in a **local** Table called “**Paste Errors**”. Click on “**OK**” to close the error box, and you are done with your pasting operation.

If there are pasting errors, the “**Paste Errors**” **local** Table is automatically **created** by MS-Access with the required number of fields. Fields will just be called “**F1**”, “**F2**”, “**F3**”... This Table is **extremely useful** because it allows you to look into **each** erroneous record to see what failed and **fix** it as required (click *L.4.2* and *L.4.3*). The “**Paste Errors**” **local** Table is **not** deleted by MS-Access, so you have to delete it manually if you so wish. However, the “**Paste Errors**” **local** Table is **overwritten** by MS-Access **every time** you do a paste records operation that produces erroneous records. This means that you should look into the “**Paste Errors**” **local** Table (or copy its information elsewhere) **before** doing another paste records action, to avoid the information it contains being lost.

Notice that if you have the “**Paste Errors**” **local** Table **open** when doing the paste, MS-Access will **not close it, nor will it create another one**. Therefore, **all the non-pasted records will be lost**. Consequently, be sure to have the “**Paste Errors**” **local** Table **closed** before pasting into a MS-Access Table.

The “**Paste Errors**” **local** Table will have the first “**n**” **fields** of the pasted cells, where “**n**” is the **minimum** between the number of columns of the copied cells and the number of columns in the cells selected for pasting. In case you are pasting as new records, then the number of cells selected for pasting is the number of cells of the Table where you are pasting.

If there are few records in the “**Paste Errors**” **local** Table, you may manually inspect them to detect the source of errors and produce the corresponding correct records. If there are lots of records, you may then copy the whole “**Paste Errors**” **local** Table, paste it in another application (e.g., Excel) and design formulas that correct all the record’s errors. Once you have produced the correct records, paste them in the Table where you are doing the paste operation.

In addition to the previous general pasting errors, you may also get there following paste-specific errors:

- “*L.4.4.1 What pasting errors can I get from invisible characters in Text fields?*”
- “*L.4.4.2 What pasting errors can I get from data copied from Excel?*”
- “*L.4.4.3 What pasting errors can I get from Windows’ cut/paste buffer?*”

#### **L.4.4.1 What pasting errors can I get from invisible characters in Text fields?**

Remind from “*E.2.2.2 How do I type-in a value in a Short Text field?*” that for **Short Text** fields MS-Access will **remove any trailing space characters** that you had **typed-in**. This is a good feature, because invisible characters in text fields are a source of errors.

However, if you **paste** a text string with **invisible** characters, the string will be **accepted as such** by MS-Access and will be **saved** as the value of the field. This is a problem, because invisible characters are a source of errors. If you want to know more about invisible character errors, you may click “*L.7.10 What apparently defective results can invisible characters produce?*”.

Also, if you **paste** a string with **invisible** characters MS-Access may do some strange things (e.g., like moving intermediate new-line characters to the end of the string). If you

want to know more about this, you may click “[L.4.4.2 What pasting errors can I get from data copied from Excel?](#)”.

Therefore, **avoid** as much as possible **pasting** strings with **invisible characters**. A good practice is to configure a field validation rule that prevents invisible characters. If you want to know more about this, you may click “[K.1.2.1 Why should I add field validation rules to my Table fields?](#)”.

Because of these errors, and other ones, I strongly advise you to follow my indications from “[E.7.4 Why should I backup my data before a bulk-change?](#)”.

#### **L.4.4.2 What pasting errors can I get from data copied from Excel?**

You may suffer **errors** when pasting data copied from Excel. I will list a few ones that I have encountered:

- If you copy from Excel 10 vertical cells, where the **first one** is **text** and the other 9 are **numbers**, into 10 numeric fields of a Table, the text value in cell number 1 will be **ignored as if it does not exist**, the numbers in Excel cells 2 to 10 will be pasted **starting in the first field**, the 10<sup>th</sup> field will **not be** modified, and **no error will be reported**. This is **not** the expected result: this may be an MS-Access **bug**.
- If you copy from Excel 10 vertical cells, where the **first two** are **text** and the other 8 are **numbers**, into 10 numeric fields of a Table, **the first two fields** will become **Null**, the numbers in Excel cells 3 to 10 will be pasted **starting in the third field**, and **no error will be reported**. You get a similar problem if the **first three** Excel cells are text and the other 7 are numbers. These are **not** the expected results: this may be an MS-Access **bug**.
- If you copy from Excel 10 vertical cells, where the **first four** are **text** and the other 6 are **numbers**, into 10 numeric fields of a Table, MS-Access will report **pasting errors**, the **first five fields** will remain **unmodified**, the numbers in Excel cells 5 to 10 will be pasted **starting in the sixth field**, and an **erroneous “Paste Errors” local Table** will be created. The “**Paste Errors**” local Table is erroneous because the **first** Excel **text** field will be **duplicated** in the **first two** records, while records 3 to 5 will correctly contain the text strings in Excel cells 2 to 4. As you may see, the “**Paste Errors**” local Table erroneously contains **one duplicated record**, with a total of **five records**. This is **not** the expected result: this may be an MS-Access **bug**.
- If you copy an Excel cell with a text string that includes one or more new-line characters and you paste it over one field value (i.e., having selected **one field** before pasting), MS-Access **will move** the new-line characters **at the end of the string**. However, if you paste it as a value (i.e., having selected **one field’s value** before pasting), MS-Access **will remove** the new-line characters from the string, enclose it between double quotes, and append all the new-line characters at the end. Notice that in **both** cases the pasted string **has been modified**.

The above errors **cannot** be fixed: this may be an MS-Access and/or an Excel **bug**. Even though this cannot be fixed, you can do a **workaround** by pasting first into a plain text editor, modify the data if needed, copy again and finally paste into MS-Access.

Because of these errors, and other ones, I strongly advise you follow my indications from “[E.7.4 Why should I backup my data before a bulk-change?](#)”.

#### L.4.4.3 What pasting errors can I get from Windows' cut/paste buffer?

The cut/paste buffer of Windows may sometimes get corrupt or do strange things. This is particularly true if you copied the **rectangle of cells** from a Navigator program or from non-Microsoft applications. If the paste operation produced strange results, this can be the cause. In that case, cancel the paste operation and redo the copy operation from the other application. If it fails again, you can restart the application and even restart the computer. If it still fails, I suggest that you try again by pasting first the data in Excel and checking what has been pasted. If there are errors, you can fix them with Excel. Once you have checked that the data in Excel is correct, copy from Excel and paste into MS-Access. If this still fails, you can paste into a plain-text editor, check the information there, and once you check is correct, copy from the plain-text editor and paste into MS-Access.

Because of these errors, and other ones, I strongly advise you to follow my indications from “E.7.4 Why should I backup my data before a bulk-change?”.

#### L.4.5 How do I fix a record I cannot delete?

If **deleting** a record, you get the error message:

*“The record cannot be deleted or changed because table 'Table\_name' includes related records.”*

this is most likely because you are trying to delete a **master** record that has **one or more** existing **slave** records in a Relationship with referential integrity that does **not** have the “**Cascade Delete Related Records**” (click D.9.5) option set. The **name** of the **slave** Table containing the **slave** records is “*Table\_name*”, from the error message above.

You **fix** this by removing all the **slave** records from Table “*Table\_name*”. If the number of **slave** records is small, you can remove them manually. If the number of **slave** records is very large (which is unusual) you may design a specific Query to delete them.

Notice that a given **master** Table may be involved in several Relationships, therefore having several **slave** Tables. If this is the case, the error message above may be shown again. You will only be able to delete the record that you wanted after you have **previously deleted all its slave records in all the Relationships** that do not have the “**Cascade Delete Related Records**” option set.

Remind that setting the “**Cascade Delete Related Records**” option may seem convenient, but it is **extremely** risky: I strongly advise you **never** set this option.

### L.5 How do I fix errors in Table/Form data?

You may click:

- “L.5.1 How do I fix a Table/Form showing “#####” in a field?”
- “L.5.2 How do I fix a Table/Form showing “#Num!”, “#Div/0!”, “#Type!” or “#Func!” in a field?”
- “L.5.3 How do I fix a Table/Form showing “#Invalid” in a field?”
- “L.5.4 How do I fix a Table/Form showing “#Deleted” in a field?”
- “L.5.5 How do I fix a Form showing “#Name?” in a field?”

- “L.5.6 How do I fix a Table/Form showing a black square in a checkbox field?”
- “L.5.7 How do I fix a Table/Form showing a wrong numeric-like value in a field?”
- “L.5.8 How do I fix a Table/Form showing a wrong Short Text value in a field?”
- “L.5.9 How do I fix a Table/Form erroneously ordering records?”
- “L.5.10 How do I fix a value not in the drop-down menu in a Table/Form field?”
- “L.5.11 How do I fix a Null in a Table/Form field configured as “Required=Yes”?”
- “L.5.12 How do I fix an apparent Null in a Table/Form Short Text field configured as “Required=Yes”?”
- “L.5.13 How do I fix an apparent zero-length string in a Table/Form field configured as “Allow Zero Length=No”?”
- “L.5.14 How do I fix field value(s) that violate(s) the field/record validation rule(s)?”
- “L.5.15 How do I fix spontaneously changing Table/Form field values?”

### **L.5.1 How do I fix a Table/Form showing “#####” in a field?**

This is most likely because the **formatted** representation of its value **does not fit** in the size of the cell in “**Datasheet View**”. What I mean is that the formatted representation of the value is **larger** than the cell’s width and height. For example, if you have the **numeric-like** value “23-02-2020”, formatted as “23 February 2020” in a field with one row of height and 10 columns of width, MS-Access will show “#####”.

This is not an error as such, so in principle you do not have to fix anything. If you want to see the value, you may do one of the following:

- Select the **field’s value** (e.g., by clicking within the field, click B.5.3). This will show the unformatted value. If the unformatted value does not fit in the field’s width and height, you can scroll right and left using the arrow keys, and MS-Access will progressively show all the digits of the unformatted value.
- Make the field’s width and/or height larger, until you can see the actual formatted value. Remind that in a **Table** or **Query result** this change will **not** be permanent unless you actually save its layout, while in a **Form** this change **will be** permanent, unless you manually undo it.
- Copy the field’s value and paste it in another application (e.g., Excel), where you can see the value.

Notice that this applies to all **numeric-like** data/field types, but not to the **Short Text** or **String** data/field types. For the case of the **Short Text** or **String** data/field types, if the field’s width and height is not enough to show the complete value of the field, MS-Access will just show the leftmost part of the text string that fits in the field’s width and height.

### L.5.2 How do I fix a Table/Form showing “#Num!”, “#Div/0!”, “#Type!” or “#Func!” in a field?

If you see one of the following in a **Table/Form** field:

**#Num!**

**#Div/0!**

**#Type!**

**#Func!**

this is most likely because this is a **Calculated** field and there has been an **error** in the **expression** that calculates its value.

If “#Num!” is shown, the expression resulted in the number-overflow “#Num!” exception-value (e.g., multiplying two numbers whose result is out of range, or dividing zero by zero, “0/0”).

If “#Div/0!” is shown, the expression resulted in the divide-by-zero “#Div/0!” exception-value (e.g., dividing non-zero by zero, “5/0”).

If “#Type” is shown, the expression resulted in the type-error “#Error” exception-value (e.g., a multiplication of a number and a text string “5\* 'text'”).

If “#Func!” is shown, the expression resulted in the table-function “#Func!” exception-value (e.g., “Log('hello’)” or “Asc(Null)”).

You **fix** this doing either or both of the following:

- Modify the field’s values of the records that are producing the error in the **Calculated** field expression.
- Modify the expression of the **Calculated** field, but you have to be **very sure** because this is **risky**.

If you want to know more about these exception-values, you may click “*J.15 What exception-value bugs can I get?*”.

Non-English MS-Access versions can show a different (translated) text. For example, the Spanish version shows “#iTipo!”, “#iNúm!” or “#iFunción!”.

### L.5.3 How do I fix a Table/Form showing “#Invalid” in a field?

If you see the following in a **Table/Form** field:

**#Invalid**

this is most likely because this is a **Calculated** field and you removed a field that was used in the expression that calculates its value.

You **fix** this by either adding the removed field to the Table or by modifying the expression of the **Calculated** field suppressing from it the removed field.

This is the table-invalid “#Invalid” exception-value. If you want to know more about exception-values, you may click “*J.15 What exception-value bugs can I get?*”.


Non-English MS-Access versions can show a different (translated) text. For example, the Spanish version shows “#Inválido”.

#### **L.5.4 How do I fix a Table/Form showing “#Deleted” in a field?**

If you see the following in a **Table/Form** field:

“#Deleted”

this is most likely because this field has been **deleted** while the Table/Form was open in an MS-Access “Object pane”. In most cases, it is not only the field that has been deleted, but the full record, and therefore all the fields of the record will show “#Deleted”. This is quite frequent when you have a Query or VBA function that removes records: if you run that Query or VBA function while the Table/Form is open, then MS-Access will show “#Deleted” in all the fields of the removed records. This can also happen if another network-user is editing the Tables.

You **fix** this by clicking on the **Refresh All**  icon from the “Home” Ribbon. This will refresh the Table/Form values shown, taking them from the currently stored Table records. All the “#Deleted” labels will then disappear. This is also fixed if you close the Table and open it again.

Non-English MS-Access versions can show a different (translated) text. For example, the Spanish version shows “#Eliminado”.

#### **L.5.5 How do I fix a Form showing “#Name?” in a field?**

If you see the following in a **Form** field:

“#Name?”

this is most likely because the origin Table field has not been found. This happens for example when you remove a field from a Table, and this Form had a field linked to the removed Table field.

You **fix** this by either removing the field from the Form or by adding (creating or renaming) the removed field to the Table.

This is the wrong-name “#Name?” exception-value. If you want to know more about exception-values, you may click “*J.15 What exception-value bugs can I get?*”.

Non-English MS-Access versions can show a different (translated) text. For example, the Spanish version shows “#¿Nombre?”.

#### **L.5.6 How do I fix a Table/Form showing a black square in a checkbox field?**

If you see **black square** in a checkbox field, this is a **deleted Yes/No** value in a field configured to be displayed as a checkbox. In this type of fields, MS-Access shows a **black square** instead of the label “#Deleted”.

This is exactly the same problem, and has the same **fix**, as described in section *L.5.4* for deleted field values.

### **L.5.7 How do I fix a Table/Form showing a wrong numeric-like value in a field?**

If you see a **numeric-like** value that is different from the actual value stored in the field, this is most likely because the **formatting** configured for this field and/or the field **width** and **height** is/are not the one you want. This is particularly frequent with **Date/Time** fields, where you can configure a format that only shows the **date-part** or the **time-part** of the stored value, and therefore, the actual stored value is not correctly shown.

You **fix** this by changing the field formatting and/or the field width and height to more suitable ones.

If you want to know more about how to change the field formatting, you may click:

- *“H.6 How do I configure the formatting of column values in a Table/Query/Form?”*

If you want to know more about how to change the field width and height, you may click:

- *“H.1 How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?”*
- *“H.2 How do I change row height, hide rows or change row order in a Table/Query/Form?”*

### **L.5.8 How do I fix a Table/Form showing a wrong Short Text value in a field?**

If you see a **Short Text** value that is different from the **actual value** stored in the field, this is most likely because the string contains invisible characters (click L.7.6) or because the field width and height is not large enough.

You **fix** this by either suppressing the invisible characters (if they are wrong) or by making the field width and/or height larger.

If you want to know how to suppress the invisible characters, you may click:

- *“E.7 How do I bulk-change my Table/Form’s data?”*

If you want to know how to make the field width and/or height larger, you may click:

- *“H.1 How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?”*
- *“H.2 How do I change row height, hide rows or change row order in a Table/Query/Form?”*

### **L.5.9 How do I fix a Table/Form erroneously ordering records?**

This is most likely because the values you see in the fields are **different from** what it is actually stored. You may check the causes for this in the previous two sections:

- *“L.5.7 How do I fix a Table/Form showing a wrong numeric-like value in a field?”*
- *“L.5.8 How do I fix a Table/Form showing a wrong Short Text value in a field?”*

This can also be because you the data type of one of the fields is different from what you think. It is very frequent to see a date or a number that is actually a **String**. Since ordering



as a *String* is different from ordering as date or as a number, the ordering you see **seems** to be wrong, but it is actually correct.

### **L.5.10 How do I fix a value not in the drop-down menu in a Table/Form field?**

If a Table contains records with values in a field that are different from the options in the field's drop-down menu (and you **do not** want this), you **fix** this this by:

- If the wrong values were entered **before** the drop-down menu was configured, you **fix** this by replacing the wrong values doing a bulk-change of your Table's data (click *E.7*).
- If the wrong values were entered **after** the drop-down menu was configured, you **fix** this by configuring the drop-down menu so it does not allow to enter values different from the ones in the menu. This implies configuring the drop-down menu property "**Limit to List=Yes**" (click *D.11.5*).

Notice that even if you configure "**Limit to List=Yes**", you will be able to **paste** records with any value, even if it is not in the menu list.

### **L.5.11 How do I fix a Null in a Table/Form field configured as "Required=Yes"?**

You **fix** this as I describe in "*E.7 How do I bulk-change my Table/Form's data?*".

If you are puzzled about how it is possible that there are **Nulls** in a field configured as "**Required=Yes**", this is because the **Nulls** were in the Table **before** the field was configured as "**Required=Yes**". If you **already** have records with **Null** in a field, and you configure the field as "**Required=Yes**", MS-Access will present you a **warning**, but the **Nulls will stay in the Table records**. If you do not want the **Nulls** to be there, you have to remove them as I have just indicated.

It is also possible that this is **not** a **Null**, and it is rather an **apparent Null**: see the next section.

### **L.5.12 How do I fix an apparent Null in a Table/Form Short Text field configured as "Required=Yes"?**

If there is an **apparent Null**, this is most likely because of one of the following causes:

- The field contains a zero-length **string**, which looks like a **Null** (an empty cell). A zero-length string is **not** a **Null** and will be correctly accepted in a field configured as "**Required=Yes**".

To **fix** this, you most likely want to configure the field as "**Allow Zero Length=No**", and then remove **all** zero-length strings (see below).

- The field contains an **invisible string** (click *L.7.8*) which looks like a **Null** (an empty cell). Notice that an invisible string is **not** a **Null** and will be correctly accepted in a field configured as "**Required=Yes**".

If you want to know more about problems caused by invisible strings, you may click "*L.7.10 What apparently defective results can invisible characters produce?*".

To **fix** this, you most likely want to introduce a field validation rule that **prevents** invisible strings (click *K.1.2.1*) and remove your invisible strings (see below).

- The field contains **several lines** of text and the **first line** of text only contains **invisible characters** (click *L.7.6*): this looks like a **Null** (an empty cell). A field contains several lines of text when its text string contains one (or more) new-line characters. Notice that this string is **not** a **Null** and will be correctly accepted in a field configured as “**Required=Yes**”.

If you want to know more about this problem, you may click “*L.7.4 How do I fix text strings and insufficient field height?*”.

To **fix** this, you most likely want to introduce a field validation rule that **prevents** invisible strings (click *K.1.2.1*) and remove your invisible strings (see below).

- If the field actually contains **Nulls**, you most likely want to remove all the **Nulls** (see below).

To remove all your zero-length strings and/or invisible strings and/or **Nulls**, you may click “*E.7 How do I bulk-change my Table/Form’s data?*”.

### **L.5.13 How do I fix an apparent zero-length string in a Table/Form field configured as “Allow Zero Length=No”?**

If there is an **apparent zero-length** string, this is most likely because of one of the following causes:

- The field contains a **Null**, which looks like a zero-length string (an empty cell). Notice that a **Null** is not a **zero-length string** and will be correctly accepted in a field configured as “**Allow Zero Length=No**”.

To **fix** this, you most likely want to configure the field as “**Required=Yes**”, and then remove all **Nulls** (see below).

- The field contains an **invisible string** (click *L.7.8*), which looks like a zero-length string (an empty cell). Notice that an invisible string is **not** a **zero-length string** and will be correctly accepted in a field configured as “**Allow Zero Length=No**”.

If you want to know more about problems caused by invisible strings, you may click “*L.7.10 What apparently defective results can invisible characters produce?*”.

To **fix** this, you most likely want to introduce a field validation rule that **prevents** invisible strings (click *K.1.2.1*) and remove your invisible strings (see below).

- The field contains **several lines** of text and the **first line** of text only contains **invisible characters** (click *L.7.6*): this looks like a **Null** (an empty cell). A field contains several lines of text when its text string contains one (or more) new-line characters. Notice that this string is **not** a **Null** and will be correctly accepted in a field configured as “**Required=Yes**”.

If you want to know more about this problem, you may click “*L.7.4 How do I fix text strings and insufficient field height?*”.

To **fix** this, you most likely want to introduce a field validation rule that **prevents** invisible strings (click *K.1.2.1*) and remove your invisible strings (see below).

- If the field actually contains **zero-length strings**, you most likely want to remove all of them (see below).

To remove all your zero-length strings and/or invisible strings and/or **Nulls**, you may click “[E.7 How do I bulk-change my Table/Form’s data?](#)”.

### **L.5.14 How do I fix field value(s) that violate(s) the field/record validation rule(s)?**

If this happens, it is because you did not fix the Table/Form data when you defined the field and/or record validation rule. If you want to know more about this, you may click “[L.2.1 How do I fix data integrity errors when saving my Table design?](#)”.

To correct all the field values that return **False** in the field validation rule and/or in the record validation rule, you may click “[E.7 How do I bulk-change my Table/Form’s data?](#)”.

### **L.5.15 How do I fix spontaneously changing Table/Form field values?**

If your Table field values are changing spontaneously by themselves, this is most likely because one of the following causes:

- There is another **simultaneous database user** that is currently changing the records. If you want to know more about simultaneous users you may click “[D.13 How do I share a database, having multiple concurrent users?](#)”.
- The changing record is a slave record in a Relationship with referential integrity and having set “**Cascade Update Related Fields**”. Remind that in this case, if you change the values of **master fields** in a **master record**, the new values will be **automatically updated** in the corresponding slave fields **in all the slave records** of this **master record**.
- You have Queries (click [F.13](#)) and/or VBA functions (click [F.13.4](#)) that modify your Table records.

None of the causes above are errors, so you will have to decide if you want to change anything or not depending on your desired database functionality.

## **L.6 How do I fix a Table/Form that I cannot open?**

You may click:

- “[L.6.1 How do I fix “The record source...does not exist”?](#)”
- “[L.6.2 How do I fix “...database engine could not find the object.”?](#)”
- “[L.6.3 How do I fix “Could not find file...”?](#)”
- “[L.6.4 How do I fix “...is not a valid path”?](#)”

### L.6.1 How do I **fix** “The record source...does not exist”?

If **opening** a **Form/Report**, you get the error message:

*“The record source 'T\_Q\_name' specified on this form or report does not exist.*

*The name of the recordsource may be misspelled, the recordsource was deleted or renamed, or the recordsource exists in a different database.*

*In the Form or Report's design view or Layout view, display the property sheet by clicking the Properties button, and then set the RecordSource property to an existing table or query.”*

this is most likely because **the Form/Report** has one (or more) fields with a **record source** Table and/or Query that could not be found.

You **fix** this by first clicking on “**OK**” to remove the error message, and then correcting the record source (click *D.10.3*).

Finally, you may alternatively want to remove this Form/Report in case it is obsolete.

### L.6.2 How do I **fix** “...database engine could not find the object.”?

If **opening** a Table/Form, you get the error message:

*“The Microsoft Access database engine could not find the object 'Table\_name'. Make sure the object exists and that you spell its name and the path name correctly. If 'Table\_name' is not a local object, check your network connection or contact the server administrator.”*

this is most likely because the **Table** is a **linked Table**, and the **source Table** 'Table\_name' **was not found in the source file**. Remind that a **linked Table** is a **link** to a Table (the **source Table**) included in **another** file (the **source file**).

You **fix** this by first clicking on “**OK**” to remove the error message, and then **relinking** the linked Table to the **correct source Table** in the source file. If you want to do this, you may click “*K.3.10 How do I view and manage Table links?*”.

Finally, you may alternatively want to remove this linked Table (and its corresponding Form, if any), but you have to be **very sure** because this **may be risky**.

### L.6.3 How do I **fix** “Could not find file...”?

If opening a Table/Form, you get the error message:

*“Could not find file 'File\_name\_with\_path'.”*

this is most likely because the **Table** is a **linked Table**, and the **source file** 'File\_name\_with\_path' **was not found**. Remind that a **linked Table** is a **link** to a Table (the **source Table**) included in **another** file (the **source file**).

You **fix** this by first clicking on “**OK**” to remove the error message, and then **relinking** the linked Table to a correct source Table in the **correct source file**. If you want to know how to do this, you may click “*K.3.10 How do I view and manage Table links?*”.

Finally, you may alternatively want to remove this linked Table (and its corresponding Form, if any), but you have to be **very sure** because this **may be risky**.

### L.6.4 How do I fix “*...is not a valid path*”?

If **opening** a Table/Form MS-Access **seems** frozen, or you get the error message:

*“File\_name\_with\_path' is not a valid path.*

*Make sure the path name is spelled correctly and that you are connected to the server on which the file resides.”*

this is most likely because the **Table** is a **linked Table**, and **the file path** to the source file “*File\_name\_with\_path*” **is wrong**. Remind that a **linked Table** is a **link** to a Table (the **source Table**) included in **another** file (the **source file**).

You **fix** this by first clicking on “**OK**” to remove the error message, and then:

- If the **source** file is in a **network drive**, and MS-Access seemed frozen (or the error message took a while to be displayed), the problem is **most likely** that the network and/or the remote server is/are down. It is therefore not possible to access the source file. You **fix** this by fixing the network connection and/or the network server.
- If the **source** file is in a **local drive** (or in a network drive currently accessible) the problem is **most likely** that the **file-path** to the source file is wrong. You **fix** this by **relinking** the linked Table to a correct source Table in a correct source file with a **correct file-path**. If you want to know how to do this, you may click “*K.3.10 How do I view and manage Table links?*”.

Finally, you may alternatively want to remove this linked Table (and its corresponding Form, if any), but you have to be **very sure** because this **may be risky**.

### L.7 How do I fix errors with Short Text or String fields?

This chapter also answers the questions:

- **How do I fix MS-Access erroneously considering two equal text strings as different text strings?**
- **How do I fix text string functions returning defective values?**
- **How do I fix text fields wrongly ordered?**
- **How do I fix an empty text field behaving as if it had data?**

A frequent source of **apparent** errors listed in these questions is that the user is **not aware** that MS-Access considers **upper**-case and **lower**-case letters to be **equal**, in **Comparison operators** (click *G.5.3*), in **record sorting** (click *F.7.12*) and in VBA operators and functions (click *J.11.22*). Once the user is aware of these facts, this source of **apparent** errors is solved.

Another frequent **source** of **apparent** errors arises from a **mismatch** between the **actual** text string contained in a Table/Query/Form field and what the database user **believes** that is the text string in the field. This **mismatch** is caused by the way text string fields are **shown**. Since the user believes that the field content is different from what it really is, the user **expects** a **different** result from what he/she is getting, although the result produced by MS-Access is **correct**.

You **fix** this by finding the **specific** problem in the sections below, and clicking on it:

- “*L.7.1 Why text strings can be different from what is shown?*”

- “L.7.2 What unexpected results can I get because text strings are different from what is shown?”
- “L.7.3 How do I fix text strings and insufficient field width?”
- “L.7.4 How do I fix text strings and insufficient field height?”
- “L.7.5 How do I fix text strings and invisible characters?”
- “L.7.6 What are invisible characters?”
- “L.7.7 What is the zero-length text string?”
- “L.7.8 What are invisible text strings?”
- “L.7.9 Why can I have invisible characters in my Short Text and String fields?”
- “L.7.10 What apparently defective results can invisible characters produce?”

### L.7.1 Why text strings can be different from what is shown?

Text strings **contained** in your Table/Query/Form fields can be **different** from what is **shown** because one or more of the following three reasons:

- **Insufficient field width**  
The field (column) **width** only shows a **given prefix** of the string, and the **remaining suffix cannot be seen**. If you want to know how to deal with this case, you may click “L.7.3 How do I fix text strings and insufficient field width?”.
- **Insufficient field height**  
The field (rows) **height** only shows the **first “n” lines** of the string, and **remaining lines cannot be seen**. If you want to know how to deal with this case, you may click “L.7.4 How do I fix text strings and insufficient field height?”.
- **Invisible characters**  
All invisible characters are **shown** as a **space**, so you **cannot visually distinguish** them. Even worse, you cannot know how many trailing invisible characters the string has, because **trailing spaces** are **not shown**. If you want to know how to deal with this case, you may click “L.7.5 How do I fix text strings and invisible characters?”.

### L.7.2 What unexpected results can I get because text strings are different from what is shown?

#### Apparent Null in Key fields

A **Short Text** Table **Key field** may **appear** to contain **Null**, because the field contents are not shown or because the field contains an invisible string or the zero-length string. This may puzzle the user, that may believe that MS-Access is not working properly or that this is **not** a **Key** field.

#### Apparent Null in “Required=Yes” fields

A **Short Text** Table **field** configured as “**Required=Yes**” may **appear** to contain **Null**, because the field contents are not shown or because the field contains an invisible string or the zero-length string. This may puzzle the user, that may believe that MS-Access is not working properly or that this is **not** a “**Required=Yes**” field.

### Apparent zero-length string in fields set as “Allow Zero Length=No”

A **Short Text** Table **field** configured as “**Allow Zero Length=No**” may **appear** to contain the **zero-length string**, because the field contents are not shown or because the field contains an invisible string. This may puzzle the user, that may believe that MS-Access is not working properly or that this field is not configured as “**Allow Zero Length=No**”.

### Apparent error in Comparison operators

Comparison operators are “=”, “<”, “>”, “<=” and “>=”. These can be applied to text strings.

The result of a Comparison operator may **appear** to be wrong because the contents of one, or both, operands are **not** completely shown or because they contain different **invisible** characters or because they contain the **zero-length** string. This may puzzle the user, that may believe that Comparison operators are not working properly.

If you want to know more about apparent errors with invisible characters, you may click “[L.7.10 What apparently defective results can invisible characters produce?](#)”.

### Apparent error in Value Sorting

The result of **value sorting** may **appear** to be wrong because some of the field value-list contents are **not** completely shown or because they contain different **invisible** characters or because they contain the **zero-length** string. This may puzzle the user, that may believe that **value sorting** is not working properly.

If you want to know more about apparent errors with invisible characters, you may click “[L.7.10 What apparently defective results can invisible characters produce?](#)”.

### Apparent error in the Length function “Len()”

The result of the “**Len()**” function may **appear** to be wrong because the contents of its argument are **not** completely shown or because it contains **invisible** characters or because it is the **zero-length** string. This may puzzle the user, that may believe that the “**Len()**” function is not working properly.

If you want to know more about apparent errors with invisible characters, you may click “[L.7.10 What apparently defective results can invisible characters produce?](#)”.

### Apparent error in *String* concatenation operators “&” and “+”

The result of the *String* concatenation operators “&” and “+” may **appear** to be wrong because the contents of one, or both, operands are **not** completely shown or because they contain different **invisible** characters or because they contain the **zero-length** string. This may puzzle the user, that may believe that these operators are not working properly.

If you want to know more about apparent errors with invisible characters, you may click “[L.7.10 What apparently defective results can invisible characters produce?](#)”.

## L.7.3 How do I fix text strings and insufficient field width?

Among the three causes of **mismatch** between the **actual** text string and the **shown** text string that are listed in “[L.7.1 Why text strings can be different from what is shown?](#)”, this one is **quite mild**. The reason is that the database user is **usually aware** that the field **width** limits the part of the text string that can be seen.

You can detect this problem in the two following ways:

- **Making the field (column) wider**  
This will show the **suffix** of the text string that is different and that was not previously shown. If you want to know how to make the field wider, you may click “*H.1.1 How do I change the column width in a Table/Query/Form?*”. Once you have checked the problem, you may discard the changes made on your field width by not saving the Table/Query/Form.
- **Selecting the field’s value**  
If you select the **field’s value** (e.g., by clicking within it, click *B.5.3*) and move along the field content pressing the right-arrow “➔” key, the text string will scroll horizontally, and you will be able to progressively see its whole contents.

In case the **actual** text string was correct, you do not modify your data, but the results you are getting **will now make sense** to you. In case the **actual** text string was **wrong**, then you should **correct** the corresponding Table field(s) and/or Query code to make it correct.

#### **L.7.4 How do I fix text strings and insufficient field height?**

Among the three causes of **mismatch** between the **actual** text string and the **shown** text string that are listed in “*L.7.1 Why text strings can be different from what is shown?*”, this one is **somehow severe**. The reason is that the database user is **not usually aware** that text strings may be **split in several lines**, and therefore, he/she **does not expect** that a string has additional lines to what is being shown.

A particularly frequent case is the text string with an **invisible first line**. This is due to the user mistakenly typing-in an initial line-feed character or pasting a field with an initial line-feed character. Since most frequently the field (row) height is one line, the field **appears** to be empty, however, **it is not**.

You can detect this problem in the two following ways:

- **Making the field (rows) higher**  
This will show **the lines** of the text string that are different and that were not previously shown. If you want to know how to make the field (rows) higher, you may click “*H.2.1 How do I change the height of all the rows in a Table/Query/Form?*”. Once you have checked the problem, you may discard the changes made on your field (rows) height by not saving the Table/Query/Form.
- **Selecting the field’s value**  
If you select the **field’s value** (e.g., by clicking within it, click *B.5.3*) and move along the field content pressing the right-arrow “➔” key, the text string will scroll horizontally, and you will be able to progressively see its whole contents, **also across different lines**.

In case the **actual** text string was correct, you do not modify your data, but the results you are getting **will now make sense** to you. In case the **actual** text string was **wrong**, then you should **correct** the corresponding Table field(s) and/or Query code to make it correct.

#### **L.7.5 How do I fix text strings and invisible characters?**

Among the three causes of **mismatch** between the **actual** text string and the **shown** text string that are listed in “*L.7.1 Why text strings can be different from what is shown?*”, this



one is **really severe**. The reason is that you **cannot** detect this problem by making the field **wider** or **higher**, nor by selecting the **field value**. No matter what you do, the strings will be **equally shown** in spite of **being different**. Particularly difficult problems are caused the **zero-length** string and by **invisible strings**.

If you want to know more about **invisible characters**, you may click “[L.7.6 What are invisible characters?](#)”. If you want to know more about **invisible strings**, you may click “[L.7.8 What are invisible text strings?](#)”.

Detecting this case is considerably more difficult than detecting the two cases presented in the previous two sections. To detect this case, you have to **copy** the suspect field and **paste** it in some **external application** (e.g., a binary text editor), that **shows** invisible characters. Using this external application, you can know for sure what is the **actual** content of the field.

In case the **actual** text string was correct, you do not modify your data, but the results you are getting **will now make sense** to you. In case the **actual** text string was **wrong**, then you should **correct** the corresponding Table field(s) and/or Query code to make it correct.

Since **detecting** this problem is quite difficult, the best approach is you **prevent** this problem. My advice is that you **always** configure **Short Text** Table fields in the following way:

- **Set “Required=Yes”**  
This prevents **Null** in your **Short Text** Table fields. If you want to know how to configure this, you may click “[D.5.1.7 What is the “Required” Table field property?](#)”.
- **Set “Allow Zero Length=No”**  
This prevents **zero-length** text strings in your **Short Text** Table fields. If you want to know how to configure this, you may click “[D.5.2.2 What is the “Allow Zero Length” Table field property?](#)”.
- **Add a field validation rule**  
Add a field validation rule that prevents invisible characters in your text fields. If you want to know more about this, you may click “[D.5.1.5 What is the field “Validation Rule” Table field property?](#)”.

## **L.7.6 What are invisible characters?**

**Invisible characters** are the ones **shown** as a **space**. The **special** characters **Space** “Chr(32)” and **no-break space** “Chr(160)” are both invisible. All **control** characters “Chr(0)” to “Chr(31)” are invisible.

**Object names** can contain invisible **special** characters (i.e., space and no-break space) but **cannot** contain invisible **control** characters. However, **text string values** in your database can contain **both** invisible **special** characters **and** some invisible **control** characters.

By far, the most frequent invisible character in text strings is **Space** “Chr(32)”, and far less frequent we can also find **Line-feed**<sup>167</sup> “Chr(10)”, horizontal **Tab** “Chr(9)” or **no-**

---

<sup>167</sup> The “line feed” character is used in text files to split the text in different lines. When a line feed character is encountered, the next character is considered to be in a different line, below the previous line.

**break space** “Chr(160)”.

Invisible characters are really bad, because they **all look like a space**, but they are **not a space**. Invisible characters in your text string values (i.e., a **Short Text** Table field or a *String* Query field) can produce **unexpected results** in operators and functions. Because they produce unexpected results in operators and functions, they will also cause **unexpected output record-lists in your Query results**.

If you press “**Ctrl-Enter**” (i.e., press the “**Ctrl**” key, and without releasing it, press the “**Enter**” key) while typing-in into a Table **Short Text** field, MS-Access will insert the **line-feed**<sup>168</sup> character. Your text string will therefore be split in two lines.

If you type-in **initial, or intermediate, spaces** and/or **new-line characters** (see the previous paragraph), within a visible text string, into a Table **Short Text** field, **all** the spaces and/or new-line characters will be preserved in the field value. Notice that with this case you can type-in **either** a visible string with invisible characters, **or**, a multiline visible string with an invisible first line.

If you type-in **trailing spaces** and/or **new-line characters**, after a visible string into a Table **Short Text** field, MS-Access will **remove all of the trailing spaces and/or new-line characters**. This is a very useful MS-Access feature to prevent unintentional errors. The field will therefore contain the string you typed-in, without all the trailing spaces or new-line characters.

If you **only** type-in **spaces and/or new-line characters** into a Table **Short Text** field, it will contain **Null** (and **not** the zero-length string, click *L.7.8*).

If you **type-in two consecutive double quotes** into a Table **Short Text** field, MS-Access will convert this to a zero-length string. The field will therefore contain a zero-length string.

If you **press** the “**Tab**” key or the “**Enter**” key into **any** Table field (including **Short Text** fields), MS-Access interprets this as if you are done entering data in the current field and will select the field to the right of the current field (or the next record if this is the rightmost field of the record).

### **L.7.7 What is the zero-length text string?**

It is a **text string** with **zero** characters.

Notice that all **invisible strings**, the **zero-length** string and a **Null** are all **equally shown** in Table/Query/Form fields as an **empty** field, but they are **all different**.

**Invisible** strings and/or **zero-length** strings in your **Short Text** Table fields or in *String* values produce **unexpected results** in operators and functions. Because they produce unexpected results in operators and functions, they will also cause **unexpected output record-lists in your Query results**.

If against my advice you want to type-in the zero-lengths string, you have to type-in **two consecutive** double quote “” characters in an empty field.

**All** invisible strings and the **zero-length** string can be **entered** in your Table fields when

---

<sup>168</sup> The “line feed” character is used in text files to split the text in different lines. When a line feed character is encountered, the next character is considered to be in a different line, below the previous line.

you **paste** values. Even if your Table fields **do not** contain any **invisible** string, nor the **zero-length** string, they can appear in your Query fields as a consequence of Query processing (click [L.7.9](#)).

### **L.7.8 What are invisible text strings?**

**Invisible** text strings are the ones that are composed of **one or more invisible characters**.

Notice that all **invisible strings**, the **zero-length** string and a **Null** are all **equally shown** in Table/Query/Form fields as an **empty** field, but they are **all different**. Also, **different invisible strings** are all **equally shown** in Table/Query/Form fields as an **empty** field, in spite of being different.

**Invisible** strings and/or **zero-length** strings in your **Short Text** Table fields or in *String* values produce **unexpected results** in operators and functions. Because they produce unexpected results in operators and functions, they will also cause **unexpected output record-lists in your Query results**.

If against my advice you want to type-in invisible characters, you may click [L.7.6](#).

or the **zero-length** string in your Table fields (click [L.4.1.6](#)). **All** invisible strings and the **zero-length** string can be **entered** in your Table fields when you **paste** values. Even if your Table fields **do not** contain any **invisible** string, nor the **zero-length** string, they can appear in your Query fields as a consequence of Query processing (click [L.7.9](#)).

### **L.7.9 Why can I have invisible characters in my Short Text and String fields?**

**Short Text** Table fields may contain invisible characters because the database user mistakenly typed them in. They can also contain invisible characters because the user mistakenly **pasted** them. In case you use Queries that **insert** Table records, this can also cause invisible characters in Table **Short Text** fields.

*String* Query fields can contain invisible characters as a consequence of expressions in the Queries that produce text strings. Even if all your Table **Short Text** fields do not contain **any invisible** character, *String* Query fields may contain them as a consequence of Query processing.

### **L.7.10 What apparently defective results can invisible characters produce?**

I now list a few **apparently defective** results that strings with invisible characters can produce.

#### **Comparison operators**

Comparison operators are "=", "<", ">", "<=" and ">=". These can be applied to text strings.

**Trailing space characters** in text strings are **ignored by all string Comparison operators**. This means that a given text string, and the same text string followed by any number of spaces produces **exactly the same result** in any comparison operator. **Other trailing invisible characters** (e.g., tabs or new-line characters) **will not be ignored** and will be taken into account by all the Comparison operators.

Therefore, with the exception of ignoring all trailing spaces, comparison operators will compare both input strings character to character, as it is done with visible characters. Recall that MS-Access is case insensitive, and therefore, any **uppercase** letter is **considered equal to** its corresponding **lowercase** letter. However, non-English characters are considered **different among themselves and from their English equivalent** (e.g., “á”, “à”, “â” and “a” are all different, and “ñ” is different from “n”).

### Value Sorting

Invisible characters are **not** equally sorted. Therefore, invisible characters may cause unexpected results when sorting on your text fields. If you want to know more about how invisible characters are sorted, you may click “[F.7.12.1 How are the different data/field types ordered by the “ORDER BY” clause?](#)”.

### Length function “Len()”

The built-in “Len()” function returns the **number of characters** of a text string.

The result of “Len(Null)” will be **Null**. The result of “Len()” over the **zero-length** string is **0**. The result of “Len()” over any other string will be the actual **number of characters** in the string. Trailing spaces are **not** ignored by the “Len()” function (as it happens with Comparison operators) and will be counted.

### Text string concatenation operators “&” and “+”

If you put **Null** as one of the two operands, it is **ignored** by the “&” and the “+” Text string operators, and the result will be the other operand.

If you put **Null** as **both** operands, the result will be **Null**.

For all the other cases, the result will be exactly the concatenation of the two **String** operands. **Trailing spaces** are **not** ignored by these operators, as it happens with Comparison operators.

## L.8 How do I fix errors with the user interface?

You may click:


- “[L.8.1 How do I fix a missing or disappearing command/tool?](#)”
- “[L.8.2 How do I fix a shaded new-record button?](#)”
- “[L.8.3 How do I fix a Query opening in “SQL View” when I selected “Design View”?](#)”
- “[L.8.4 How do I fix the “Access SQL Editor” marking a Query as having unsaved changes?](#)”
- “[L.8.5 How do I fix an error when opening a Query in “Design View”?](#)”
- “[L.8.6 How do I fix an extra column in my Tables in “Datasheet View”?](#)”
- “[L.8.7 How do I fix an extra row in my Query results?](#)”
- “[L.8.8 How do I fix the database users modifying Table values by editing Query results?](#)”
- “[L.8.9 How do I fix missing columns/rows in my Query results?](#)”

- “L.8.10 How do I fix “Could not find field...” when doing “Compact and Repair” or “Save As”? How do I fix “Could not find field...” when doing “Compact and Repair” or “Save As”?”
- “L.8.11 How do I fix the VBA editor changing the value of the constants I write?”
- “L.8.12 How do I fix foreign-language issues of MS-Access?”

### L.8.1 How do I fix a missing or disappearing command/tool?

The tools and commands shown by MS-Access are **automatically changed** for different reasons. This can be very puzzling if you are not aware of it. If you want more to know more about this, you may click “B.3.2 Why do Ribbons change how they look?”.

### L.8.2 How do I fix a shaded new-record button?

If the **local database file** of a Table, or the **source file** (click K.3.3) for a linked Table (K.3.4), is **read-only**, the “new-record” button “

You **fix** this by **enabling write** permission in the corresponding file(s).

### L.8.3 How do I fix a Query opening in “SQL View” when I selected “Design View”?

If your Query is a **Union operation** (i.e., the outermost operation is a **Union**), MS-Access will **not allow** you to open it in “**Design View**” (click B.4.1.3). If you **try** to do it, the Query will **always** be opened in “**SQL View**”. Likewise, if you try to change its **view-type** (click B.4.1.4), the option “**Design View**” will **not** be shown. You **fix** this by **enclosing** the outermost **Union** operation of the Query in a **Select** operation. Recall also to **qualify all** its **output field names** to avoid losing all the column formatting of the Query (click K.4.5) after having changed its SQL code. Enclosing the outermost **Union** in a **Select** is a **good practice** because it allows you to configure the formatting of your fields, ordering the records using the “**ORDER BY**” clause, as well as other advantages.

If your Query is **not a Union operation**, MS-Access may **sometimes** open it “**SQL View**”, even if you selected “**Design View**” in the “**Navigation Pane**” pop-up menu. This may be an MS-Access **bug**. You **fix** this by changing the Query’s **view-type** to “**Design View**” (click B.4.1.4).

### L.8.4 How do I fix the “Access SQL Editor” marking a Query as having unsaved changes?

If you modify a Query’s SQL code in “**SQL view**” and/or in “**Design view**”, its SQL code in the “**Access SQL Editor**” will **not** be modified, so there will be **two different** SQL codes for the Query: the SQL code in MS-Access and the SQL code in the “**Access SQL Editor**”.

For this reason, when you open the Query in the “**Access SQL Editor**”, the editor will display the SQL code that it stores, but will mark it as containing **unsaved changes**, to indicate that the code is different from the one stored in MS-Access.

If you **save** changes, the SQL code of the Query in MS-Access will become the one from the “**Access SQL Editor**”, and the changes you did in “**SQL view**” and/or “**Design view**”

will be overwritten.

If you **do not save** changes, the SQL code of the Query in MS-Access will **remain** the same (including the changes you did in “**SQL view**” and/or “**Design view**”), and the code in the “**Access SQL Editor**” will **also remain** the same (but different to the one in MS-Access). In this case, the situation of having **two different** SQL codes for the Query will continue.

While the situation of having **two different** SQL codes stored for the Query persists, every time you open the Query in the “**Access SQL Editor**” it will be **marked** as containing **unsaved changes** (with an asterisk on the left of its name). In this way, the “**Access SQL Editor**” indicates that its stored SQL code is different from the one stored in MS-Access.

If you **run** the Query from MS-Access, the result will correspond to the SQL code stored in MS-Access. If you **run** the Query from the “**Access SQL Editor**”, you have two cases:

- If you **ticked** the recommended option “Automatically save queries before running” (click *F.5.1.3*), the Query’s SQL code is automatically **saved** when you **run** the Query. Therefore, the Query’s SQL code in MS-Access becomes the one that was stored in the “**Access SQL Editor**”. Consequently, there is no difference in SQL codes, and the result of running the Query will correspond to the SQL code that was stored in the “**Access SQL Editor**”.
- If you did not **tick** that option, the “**Access SQL Editor**” will **not** run the Query unless you **first** save its unsaved changes.

Finally, notice that you **unwillingly** modify a Query’s SQL code **just by opening it** in “**Design View**”, in the case that MS-Access **cannot** represent the Query’s SQL code in its semi-graphical interface of “**Design View**”. If you want to know more about this, you may click “*L.8.5 How do I fix an error when opening a Query in “Design View”?*”.

### **L.8.5 How do I fix an error when opening a Query in “Design View”?**

If **opening** a Query in “**Design View**”, or **changing** a Query’s view-type to “**Design View**”, you get the error message:

***“Microsoft Access can't represent the join expression on\_exp() in Design view.***

*\* One or more fields may have been deleted or renamed.*

*\* The name of one or more fields or tables specified in the join expression may be misspelled.*

*\* The join may use an operator that isn't supported in Design view, such as > or <.”*

with a single “**OK**” button. And if you close this error message-box (either clicking on “**OK**” or on its close “**X**” icon), you may get a few more similar ones.

This is most likely because you wrote an SQL that MS-Access cannot represent with its semi-graphical interface in Query “**Design View**”. See further below for more detail.

You **fix** this by:

1. Do whatever formatting changes you wanted to do in “**Design View**”.
2. **Save** the Query design (click *B.4.1.6*) and **close** the Query (click *B.4.1.7*).

3. **Open** the Query code in the “**Access SQL editor**” (click *F.5.4.1*).
4. **Save** the Query code (click *F.5.4.6*) and **close** it (click *F.5.4.7*), from the “**Access SQL editor**”.

The reason for the error above is that MS-Access has severe limitations in the SQL code that it can represent with its semi-graphical interface in Query “**Design View**”. Consequently, when you open one of your Queries in “**Design View**”, it is not infrequent that MS-Access will show the error message above. The problem is that MS-Access **will modify the SQL code of your Query**, to make it comply with which it can represent in “**Design View**”. Therefore, after you save the design changes you did in the Query “**Design View**”, your SQL code **will be different!** Luckily, the changes that MS-Access does on the SQL code **do not** affect your SQL code stored in the “**Access SQL editor**”. This is why you have to do steps **3** and **4** above, in order to restore your SQL code, overwriting the changes done by MS-Access.

### **L.8.6 How do I fix an extra column in my Tables in “Datasheet View”?**

You **fix** this by clicking the sequence **File**→**Options**→**Current Database** and then, under the heading “**Application Options**”, untick the checkbox for “**Enable design changes for tables in Datasheet view**”.

This will prevent MS-Access from showing the column “Click to Add” in your Tables. I consider this column very distracting, and I also think it is **very risky**, because the database user may unwillingly add a field to the Table by mistake.

### **L.8.7 How do I fix an extra row in my Query results?**

You may see an **extra row at the end** your Query results for the following causes:

- **Row to add Table record (the new-record row)**  
Having the **new-record row** in your Query results is **extremely risky**. My advice is you **always prevent** it from appearing in your Queries. If you want to know more, you may click “*K.4.4 Why should I disable changing Table data from Query results?*”.
- **Row labeled Total in its leftmost cell**  
You remove this row clicking on the **Total** “ $\Sigma$ ” icon from the “**Home**” Ribbon. If you want to know more, you may click “*H.8 How do I show aggregate values (e.g., totals) in a Table/Query/Form?*”.

### **L.8.8 How do I fix the database users modifying Table values by editing Query results?**

You **fix** this by preventing the users from modifying Table values from Query results: click “*K.4.4 Why should I disable changing Table data from Query results?*”.

### **L.8.9 How do I fix missing columns/rows in my Query results?**

You **fix** this by showing the hidden columns and/or modifying the filters. To know more about this, you may click:

- “*H.4 How do I hide/unhide columns in a Table/Query/Form?How do I hide/unhide*

columns in a Table/Query/Form?”

- “*H.2.2 How do I hide rows in a Table/Query/Form?How do I hide rows in a Table/Query/Form?”*

### **L.8.10 How do I fix “Could not find field...” when doing “Compact and Repair” or “Save As”?**

If clicking “**Compact and Repair Database**” (click *E.10*) or doing “**Save As**” of your database file, you get the error message:

“*Could not find field 'Field\_name'.*”

this is most likely because you removed a field ‘*Field\_name*’ that was used in a **Calculated** field and/or in the Table’s record validation rule.

You **fix** this by **locating** and **opening** the Table and doing either of the following:

- **Correct** the corresponding **Calculated** field(s) and/or the Table’s record validation rule so they refer to existing field names.
- **Add** a field (creating or renaming it) with the same field name as the missing one.
- **Remove** the corresponding **Calculated** field(s) and/or Table’s record validation rule. This is **risky**, so you have to be very sure that this is what you want.

To **locate which Table** produced the error notice that:

- You recently **removed** the field “*Field\_name*” from it.
- It has **Calculated** fields and/or a record validation rule.

Notice that MS-Access **will not allow you** to do the “**Save As**” of your database file until you have fixed this. In case you want a copy of your file **now**, without fixing first the error, a workaround is saving and closing the database file, copying the file, and then change the file name of the copy to the one you wanted to use in the “**Save As**” operation.

### **L.8.11 How do I fix the VBA editor changing the value of the constants I write?**

You **fix** this by either changing the data type or the constant writing format.

For changing the data type, you may click “*L.4.1.3 How do I fix a changed numeric value?*”.

For the constant writing format, you may click “*G.4 How do I write a constant?*”.

### **L.8.12 How do I fix foreign-language issues of MS-Access?**

If you are using a foreign-language version of MS-Access (Spanish, French, German, ...) you may encounter some differences from what is described in this Lightning Guide, that I am summarizing here:

#### **Values shown in “Datasheet View”**

- In “**Datasheet View**”, the **values shown** have **translated** thousands/decimal **separators**, month **names**, month **order** and **Boolean** values. For example, in the Spanish version you see “**5.435,23**” instead of “**5,435.23**”, “**23-Abril-2021**” instead



of “April-23-2021” and “Activado” instead of “On”.

### Entering values and writing constants

- In **SQL expressions** and **VBA expressions**, **constants** are written like the **English** version, regardless of the language version. You write **Null**, **True**, **Yes** or **On**. *Date* constants are **interpreted** month first. Decimal separator is the period “.” character.
- In “**Datasheet View**” and in **Table “Design View” expressions**, to write a *Boolean* value (either entering a value or writing a constant) you type-in the **translated Boolean** value. For example, in the Spanish version you type-in “**Verdadero**”, “**Sí**” or “**Activado**”, instead of “**True**”, “**Yes**” or “**On**” (respectively).
- In “**Datasheet View**” and in **Table “Design View” expressions**, to write a **fractional** value (either entering a value or writing a constant) you type the **translated decimal separator**. For example, in the Spanish version you would type-in “**5,3**” instead of “**5.3**”.
- In “**Datasheet View**” and in **Table “Design View” expressions**, to write the **name** of a month (either entering a value or writing a constant) you type-in the **translated month name**. For the specific case of “**Datasheet View**” you can **also** type-in **any prefix of the English month name**, down to its three first letters.
- In **Table “Design View” expressions**, to write a **Null** “constant” you write the translated name. For example, in the Spanish version you write “**Nulo**”.

### Operator names and functions names

- In **Table “Design View” expressions**, **most** (but **not all**) operator names (e.g. “**AND**”, “**OR**”, ...) and built-in function names (e.g., “**Left()**”) **are translated** to the foreign language. For example, in the Spanish version the said operators and function name become, “**Y**”, “**O**” and “**Izq()**”. If you want more information about this, you may click “*G.1 What are the main differences between the three expression scopes?*”.

### CStr()” type conversion function

The “**CStr()**” type-conversion function behaves differently depending on the data type of its argument, as follows:

- **CStr()** over *Boolean*:  
Works like the **English** version, regardless of the language version. It produces the *String* “-1” for *True* and the *String* “0” for *False*. Therefore, this works the same regardless of the language version.
- **CStr()** over **fractional number**:  
Produces a *String* using the **foreign-language decimal separator**. For example, in the Spanish version “**Cstr(4.5)**” produces the *String* “**4,5**”.
- **CStr()** over *Date*:  
Produces a *String* using two-digit **numeric month** (not month name) with the **foreign-language month ordering** and separated by “/”. For example, in the Spanish version “**Cstr(#Mar-28-2021#)**” produces the *String* “**28/03/2021**”, because in Spanish the month goes after the day.

**“PIVOT” field names**

- **From *Boolean* values**  
Works like the **English** version, regardless of the language version. The value *True* produces the “**PIVOT**” **field name** “-1” and the value *False* produces the “**PIVOT**” **field name** “0”.
- **From fractional number constants**  
Produces the “**PIVOT**” **field name** using the **foreign-language decimal separator**. For example, in the Spanish version the **constant** value “4.5” produces the “**PIVOT**” **field name** “4,5”.
- **From fractional number expressions**  
Works like the **English** version, regardless of the language version. The decimal separator is always converted to the underscore “\_” character. For example, in the Spanish version the **expression result** “3,5” (which is “3.5” in English) is converted to the “**PIVOT**” **field name** “3\_5”. This may be an MS-Access **bug**. Notice that you can fix this by enclosing the **expression** inside the “CStr()” type-conversion function, that would produce the “**PIVOT**” **field name** “3,5”.
- **From *Date* values**  
The **month order** is the one of the foreign language. For example, if the “**PIVOT**” **expression** produces the zero-time *Date* value “March 28, 2021”, the Spanish version will produce the “**PIVOT**” **field name** “28/03/2021”, because in Spanish the day goes before the month.

**Various issues**

- **Error and warning messages** are **translated** to the foreign language.
- In **SQL code**, the **SQL keywords** are the ones of the **English** version, regardless of the language version. Therefore, you always write the keywords (“**SELECT**”, “**FROM**”, “**WHERE**”, ...) stated in this **Lightning Guide**.
- In **custom formatting** of fields, a few format-control characters are **translated** (e.g., in the Spanish version “**y**” for year becomes “**a**” for “año”). If you want more information about this, you may click “[H.6.2.3 How do I configure custom formatting for Number, Large Number and Currency fields?](#)”.
- Exception-value **names** are **translated**. For example, in the Spanish version “**#Num!**” becomes “**#jNúm!**”; in the French version “**#Error**” becomes “**#Erreur**”. If you want more information about exception-values, you may click “[L.5.2 How do I fix a Table/Form showing “#Num!”, “#Div/0!”, “#Type!” or “#Func!” in a field?](#)” and subsequent sections.
- Some **variable names** in the Query SQL code will be **translated** to the foreign-language!! Furthermore, if you are using the “**Access SQL Editor**”, the SQL code in the “**Access SQL Editor**” will remain **unchanged**, and if you check the Query code you will still see the English field names there. This may be an extremely puzzling problem if you are not aware of it. This problem happens in foreign-language versions of MS-Access when you are using a variable (e.g., a Table field) name that has an English meaning like “Quarter”. If you have a Query that uses field name

“Quarter”, and you modify the Query design in “**Design View**” (for example selecting ascending order), MS-Access will **replace** in the SQL code the field name “Quarter” by the MS-Access translated name (in Spanish, it is replaced by “Trimestre”). When you then run the Query, it will request parameter “Trimestre”, because it is a field name that is used in the Query (because MS-Access **changed** it) but it **does not exist** in the Table. I did a few trials and so far I have seen that this affects English variable names “Year”, “Quarter”, “Month”, “Day” and “Trim”, that are all translated to the foreign-language. This may be an MS-Access **bug**. You can fix as indicated in *L.8.5*.

- Field **types-sizes** in Table “**Design View**” are **translated** to the foreign-language. For example, in the Spanish version the field types **Yes/No**, **Number** and **Date/Time** become **Si/No**, **Número** and **Fecha/Hora** (respectively).

Along this section, I mentioned several times the **Table “Design View” expressions**. Just for completeness, remind that these **expressions** are the ones in **Calculated** fields, in record/field validation rules, in field default value and in the “**Lookup**” property.

## PART M. LIST OF BUILT-IN FUNCTIONS

This is the complete listing of built-in functions that you can use in your VBA expressions and/or in your SQL expressions. Some of these functions can be used both in your VBA and your SQL expression and some are specific of either VBA or SQL expressions. In case a function cannot be used in your VBA or in your SQL expressions I will indicate explicitly.

Remind that the built-in functions that you can use in **Table “Design View”** expressions (record/field validation rule, default value, calculated field and “**Lookup**” property) use a **semicolon “;”** character as **argument separator** and in foreign-language versions of MS-Access their names will be **translated**. If you are using a foreign-language version of MS-Access, you may click “*L.8.12 How do I fix foreign-language issues of MS-Access?*”.

This list of functions has been copied from the official Microsoft support site [support.office.com](https://support.office.com), and each function below has a web link to the function explanation in the Microsoft support web. Doing click<sup>169</sup> on each of the functions below will take you to the official Microsoft support web page corresponding to the function you clicked on. You may get direct access to the official Microsoft support web page on MS-Access functions in the following link:

<https://support.office.com/en-us/article/access-functions-by-category-b8b136c3-2716-4d39-94a2-658ce330ed83>

You may click:

- “*M.1 ActiveX functions*”
- “*M.2 Application functions*”
- “*M.3 Array functions*”
- “*M.4 Conversion functions*”
- “*M.5 Database functions*”
- “*M.6 Date and Time functions*”
- “*M.7 Domain Aggregate functions*”
- “*M.8 Error Handling functions*”
- “*M.9 File Input/Output functions*”
- “*M.10 Financial functions*”
- “*M.11 Inspection functions*”
- “*M.12 Mathematical functions*”
- “*M.13 Message functions*”
- “*M.14 Miscellaneous functions*”

---

<sup>169</sup> Notice that the web pages in the Microsoft support site may have changed since this book was released, and these links may not work.

- “M.15 Program Flow functions”
- “M.16 SQL aggregate functions”
- “M.17 File Management functions”
- “M.18 Text Processing functions”

## M.1 ActiveX functions

**ActiveX** functions **cannot** be used in **SQL** expressions.

[CreateObject\(\) function](#)

[GetObject\(\) function](#)

## M.2 Application functions

**Application** functions **cannot** be used in **SQL** expressions.

[Command\(\) function](#)

[Shell\(\) function](#)

## M.3 Array functions

**Array** functions **cannot** be used in **SQL** expressions.

[Array\(\) function](#)

[Filter\(\) function](#)

[Join\(\) function](#)

[LBound\(\) function](#)

[Split\(\) function](#)

[UBound\(\) function](#)

## M.4 Conversion functions

The type conversion functions **CDec()** and **CLngLng()** **cannot** be used in **SQL** expressions.

[Asc\(\) function](#)

[Chr\(\) function](#)

[EuroConvert\(\) function](#)

[FormatCurrency\(\) function](#)

[FormatDateTime\(\) function](#)

[FormatNumber\(\) function](#)

[FormatPercent\(\) function](#)

[GUIDFromString\(\) function](#)

[Hex\(\) function](#)

[Nz\(\) function](#)

[Oct\(\) function](#)

[Str\(\) function](#)

[StringFromGUID\(\) function](#)

[Type Conversion functions CBool\(\), CByte\(\), CCur\(\), CDate\(\), CDbl\(\), CDec\(\), CInt\(\),](#)

[CLng\(\), CLngLng\(\), CSng\(\), CStr\(\), and CVar\(\)  
Val\(\) function](#)

## M.5 Database functions

**Database** functions **DDE()**, **DDEInitiate()**, **DDERequest()** and **DDESend()** cannot be used in **SQL** expressions.

[DDE\(\) function](#)

[DDEInitiate\(\) function](#)

[DDERequest\(\) function](#)

[DDESend\(\) function](#)

[Eval\(\) function](#)

[Partition\(\) function](#)

## M.6 Date and Time functions

**Date and time** functions can be used both in **VBA** expressions and in **SQL** expressions.

[Date\(\) function](#)

[DateAdd\(\) function](#)

[DateDiff\(\) function](#)

[DatePart\(\) function](#)

[DateSerial\(\) function](#)

[DateValue\(\) function](#)

[Day\(\) function](#)

[Hour\(\) function](#)

[Minute\(\) function](#)

[Month\(\) function](#)

[MonthName\(\) function](#)

[Now\(\) function](#)

[Second\(\) function](#)

[Time\(\) function](#)

[Timer\(\) function](#)

[TimeSerial\(\) function](#)

[TimeValue\(\) function](#)

[Weekday\(\) function](#)

[WeekdayName\(\) function](#)

[Year\(\) function](#)

## M.7 Domain Aggregate functions

**Domain Aggregate** functions can be used both in **VBA** expressions and in **SQL** expressions.

[DAvg\(\) function](#)

[DCount\(\) function](#)

[DFirst\(\), DLast\(\) functions](#)

[DLookup\(\) function](#)

[DMin\(\), DMax\(\) functions](#)

[DStDev\(\), DStDevP\(\) functions](#)

[DSum\(\) function](#)

[DVar\(\), DVarP\(\) functions](#)

## M.8 Error Handling functions

function **CVErr()** cannot be used in **SQL** expressions.

[CVErr\(\) function](#)

[Error\(\) function](#)

## M.9 File Input/Output functions

**File Input/Output** functions cannot be used in **SQL** expressions.

[EOF\(\) function](#)

[FreeFile\(\) function](#)

[Input\(\) function](#)

[Loc\(\) function](#)

[LOF\(\) function](#)

[Seek\(\) function](#)

## M.10 Financial functions

Financial functions **IRR()** and **NPV()** cannot be used in **SQL** expressions.

[DDB\(\) function](#)

[FV\(\) function](#)

[IPmt\(\) function](#)

[IRR\(\) function](#)

[MIRR\(\) function](#)

[NPer\(\) function](#)

[NPV\(\) function](#)

[Pmt\(\) function](#)

[PPmt\(\) function](#)

[PV\(\) function](#)

[Rate\(\) function](#)

[SLN\(\) function](#)

[SYD\(\) function](#)

## M.11 Inspection functions

Inspection functions **Environ()**, **GetAllSettings()**, **GetSetting()** and **IsArray()** cannot be used in **SQL** expressions.

[Environ\(\) function](#)

[GetAllSettings\(\) function](#)

[GetSetting\(\) function](#)

[IsArray\(\) function](#)

[IsDate\(\) function](#)

[IsEmpty\(\) function](#)  
[IsError\(\) function](#)  
[IsMissing\(\) function](#)  
[IsNull\(\) function](#)  
[IsNumeric\(\) function](#)  
[IsObject\(\) function](#)  
[TypeName\(\) function](#)  
[VarType\(\) function](#)

## M.12 Mathematical functions

**Mathematical** functions can be used both in **VBA** expressions and in **SQL** expressions.

[Abs\(\) function](#)  
[Atn\(\) function](#)  
[Cos\(\) function](#)  
[Exp\(\) function](#)  
[Int\(\), Fix\(\) functions](#)  
[Log\(\) function](#)  
[Rnd\(\) function](#)  
[Round\(\) function](#)  
[Sgn\(\) function](#)  
[Sin\(\) function](#)  
[Sqr\(\) function](#)  
[Tan\(\) function](#)

## M.13 Message functions

**Message** functions **cannot** be used in **SQL** expressions.

[InputBox\(\) function](#)  
[MsgBox\(\) function](#)

## M.14 Miscellaneous functions

functions **CallByName()**, **Spc()** and **Tab()** **cannot** be used in **SQL** expressions. function **IMEStatus()** is specific of East Asia versions. functions **MacID()** and **MacScript()** are specific for Mac computers.

[CallByName\(\) function](#)  
[IMEStatus\(\) function](#)  
[MacID\(\) function](#)  
[MacScript\(\) function](#)  
[QBColor\(\) function](#)  
[RGB\(\) function](#)  
[Spc\(\) function](#)  
[Tab\(\) function](#)



## M.15 Program Flow functions

**DoEvents()** function **cannot** be used in **SQL** expressions.

[Choose\(\) function](#)

[DoEvents\(\) function](#)

[IIf\(\) function](#)

[Switch\(\) function](#)

## M.16 SQL aggregate functions

**SQL aggregate functions cannot** be used in **VBA** expressions. They can only be used in **SQL** expressions, and only in **some** expressions belonging to a **Select-group\_by\_aggreg**, to a **Select-total\_aggreg** or to a **Transform** operation.

[Avg\(\) function](#)

[Count\(\) function](#)

[Min\(\), Max\(\) functions](#)

[StDev\(\), StDevP\(\) functions](#)

[Sum\(\) function](#)

[Var\(\), VarP\(\) functions](#)

## M.17 File Management functions

**File Management** functions **cannot** be used in **SQL** expressions.

[CurDir\(\) function](#)

[Dir\(\) function](#)

[FileAttr\(\) function](#)

[FileDateTime\(\) function](#)

[FileLen\(\) function](#)

[GetAttr\(\) function](#)

## M.18 Text Processing functions

**Text Processing** functions can be used both in **VBA** expressions and in **SQL** expressions.

[Format\(\) function](#)

[InStr\(\) function](#)

[InStrRev\(\) function](#)

[LCase\(\) function](#)

[Left\(\) function](#)

[Len\(\) function](#)

[LTrim\(\), RTrim\(\), and Trim\(\) functions](#)

[Mid\(\) function](#)

[Replace\(\) function](#)

[Right\(\) function](#)

[Space\(\) function](#)

[StrComp\(\) function](#)

[StrConv\(\) function](#)

[String\(\) function](#)

[StrReverse\(\) function](#)

[UCase\(\) function](#)

## PART N. CONTENTS AND ACKNOWLEDGEMENTS

### Acknowledgements

I would like to thank my wife Marifeli Sedano-Ruiz, my daughter Maite Azcorra-Sedano and my son Jaime Azcorra-Sedano for their support and comprehension, along countless weekends and vacation days that I devoted to working on this book. They also designed the book cover and helped me with proofreading and catching errors and mistakes.

I would also like to thank all the members of my research group at University Carlos III of Madrid and IMDEA Networks Institute for their feedback and useful suggestions along the years that took me writing this book: Prof. Marcelo Bagnulo-Braun, Prof. Albert Banchs-Roca, Prof. Carlos Jesús Bernardos-Cano, Prof. Ruben Cuevas Rumín, Prof. Antonio de la Oliva Delgado, Prof. Alberto García-Martínez, Prof. Carmen Guerrero-López, Prof. Marco Gramaglia, Prof. Pablo Serrano-Yáñez-Mingot, Prof. Francisco Valera-Pintor and Prof. Ivan Vidal-Fernández.

Finally, thanks to our support engineer Jonathan Almodovar-Herrerros, who did the web page and document download design.

### Table of Contents

PART A. CREATING MY FIRST DATABASE WITH MS-ACCESS .....	1
A.1 How do I use this Lightning Guide?.....	1
A.2 What version of MS-Access is this Guide for? .....	2
A.3 How do I create my first database?.....	2
A.3.1 What is the MS-Access user interface? .....	2
A.3.2 How do I create my first database file? .....	2
A.3.3 How do I create my first Table? .....	3
A.3.4 How do I input my first data into my Table?.....	5
A.4 How do I write and run my first SQL Query? .....	5
A.4.1 How do I write my first SQL Query? .....	6
A.4.2 How do I run my first SQL Query? .....	7
A.4.3 What does it mean “case insensitive”? .....	7
A.5 How do I add a Table to my first database? .....	8
A.5.1 How do I add the Table “T_Capital_Rainfall_Q” to my first database?.....	8
A.5.2 How do I input data into the Table “T_Capital_Rainfall_Q”?.....	8
A.6 How do I write and run my first Select Query with record aggregation?.....	9
A.6.1 How do I write my first Select Query with record aggregation?.....	9
A.6.2 How do I run my first Select Query with record aggregation?.....	10
A.7 How do I configure my Tables in my first database?.....	11
A.7.1 How do I configure my Table “T_Capital_Cities”? .....	11
A.7.1.1 How do I configure the Primary Key field in Table “T_Capital_Cities”? .....	11

A.7.1.2	How do I configure no zero-length in all my text fields?.....	12
A.7.1.3	How do I check that the “Required” property really works? .....	12
A.7.1.4	How do I check that the Key field really works? .....	13
A.7.2	How do I configure my Table “T_Capital_Rainfall_Q”? .....	13
A.7.2.1	How do I configure the Primary Key fields in Table “T_Capital_Rainfall_Q”? .....	14
A.7.2.2	How do I configure “Allow Zero Length=No” in all my text fields? .....	15
A.7.2.3	How do I configure a field validation rule in my “Quart” field? .....	15
A.7.2.4	How do I check that the composite Key fields really work? .....	15
A.7.2.5	How do I check that the field validation rule for the field “Quart” really works? .....	16
A.8	How do I write and run my first Union Query?.....	16
A.8.1	How do I write my first Union Query?.....	16
A.8.2	How do I run my first Union Query?.....	18
A.9	How do I create a Relationship in my first database? .....	18
A.9.1	How do I create a Relationship from “T_Capital_Cities” to “T_Capital_Rainfall_Q”? .....	18
A.9.2	How do I check that the Relationship really works? .....	19
A.10	How do I write and run my first Join Query?.....	20
A.10.1	How do I write my first Join Query? .....	20
A.10.2	How do I run my first Join Query?.....	21
A.11	How do I write and run my first Transform Query?.....	21
A.11.1	How do I write my first Transform Query?.....	21
A.11.2	How do I run my first Transform Query?.....	23
PART B.	BRIEFING ON MS-ACCESS USER INTERFACE.....	24
B.1	What options should I set in MS-Access? .....	24
B.1.1	What “Current Database” options should I set?.....	25
B.1.2	What “Object Designers” options should I set?.....	26
B.1.3	What “Client Settings” options should I set? .....	26
B.1.4	What “Quick Access Toolbar” options should I set? .....	26
B.2	How is the MS-Access window structured? .....	27
B.2.1	What is the top window frame? .....	28
B.2.2	What is the “Quick Access Toolbar”?.....	28
B.2.3	What is the “Ribbon-bar”?.....	28
B.2.4	What are Ribbon names? .....	28
B.2.5	What is the Ribbon Area? .....	29
B.2.6	What is the visible Ribbon? .....	29

B.2.7	What is the “Navigation Pane”?	29
B.2.8	What is the Object Area?	30
B.2.9	What are the object tabs?	30
B.2.10	What is the Visible Object and the object panes?	30
B.2.11	What is the “Status-bar”?	31
B.3	What are the Ribbons?	31
B.3.1	What is a Ribbon group?	31
B.3.2	Why do Ribbons change how they look?	32
B.3.3	What permanent Ribbon can I display?	33
B.3.4	What contextual Ribbon can I display?	34
B.3.4.1	What are contextual Ribbons?	34
B.3.4.2	What are contextual Ribbon labels?	34
B.3.4.3	What are contextual Ribbon names?	34
B.3.4.4	What contextual Ribbon can I display, depending on context?	34
B.4	What is the “Navigation Pane” and how it works?	36
B.4.1	How do I open, close and do other actions on objects in the “Navigation Pane”?	38
B.4.1.1	How do I search for an object name in the “Navigation Pane”?	38
B.4.1.2	What is the object menu in the “Navigation Pane”?	38
B.4.1.3	How do I open an object with the “Navigation Pane”?	41
B.4.1.4	How do I change the view-type of an opened object?	41
B.4.1.5	How do I view another of the opened objects?	43
B.4.1.6	How do I save the layout/design of an opened object?	43
B.4.1.7	How do I close an opened object?	43
B.4.1.8	How do I rename an object?	44
B.4.1.9	How do I run a Query?	44
B.4.1.10	How do I select one or more objects?	45
B.4.1.11	How do I edit a Query?	45
B.4.1.12	How do I create an object?	45
B.4.1.13	How do I delete objects?	46
B.4.1.14	How do I copy/cut and paste objects?	46
B.4.2	How do I configure the way object names are grouped, ordered, displayed or hidden in the “Navigation Pane”?	46
B.4.2.1	How do I configure the way object names are grouped in the “Navigation Pane”?	47
B.4.2.2	How do I configure the way object names are ordered in the “Navigation Pane”?	48

B.4.2.3	How do I configure the way object names are displayed in the “Navigation Pane”?	49
B.4.2.4	How do I hide/unhide whole object name groups in the “Navigation Pane”?	49
B.4.2.5	How do I hide/unhide an individual object name in its object group?	50
B.4.2.6	How do I hide/unhide all system object names?	51
B.5	What is a Table/Query/Form in “Datasheet View”?	51
B.5.1	What is the “Navigation bar” in “Datasheet View”?	53
B.5.2	How do I select a range of fields/records in “Datasheet View”?	53
B.5.3	How do I select one field’s value in “Datasheet View”?	57
B.5.4	What are the differences between selecting one field and one field’s value in “Datasheet View”?	59
B.5.5	How do I find a record in a Table/Query/Form “Datasheet View”?	60
B.5.6	How do I manage a Table/Query/Form “Property Sheet” in “Datasheet View”?	60
B.6	What is a Table in “Design View”?	60
B.6.1	How do I view, add, delete and configure my Table fields?	62
B.6.1.1	How do I view the Table fields in “Design View”?	62
B.6.1.2	What are a Field’s “General” tab properties in “Design View”?	63
B.6.1.3	What are a Field’s “Lookup” tab properties in “Design View”?	63
B.6.1.4	How do I select one or more Table fields in “Design View”?	63
B.6.1.5	How do I add Table fields in “Design View”?	64
B.6.1.6	How do I copy/cut and paste Table fields in “Design View”?	65
B.6.1.7	How do I reconfigure my Table fields in “Design View”?	66
B.6.1.8	How do I delete Table fields in “Design View”?	66
B.6.1.9	How do I reorder Table fields in “Design View”?	67
B.6.2	How do I manage Table indexes in “Design View”?	67
B.6.2.1	How do I view Table indexes in “Design View”?	67
B.6.2.2	How do I select fields in Table indexes in “Design View”?	69
B.6.2.3	How do I delete, insert or move fields in Table indexes in “Design View”?	70
B.6.3	How do I unhide/show a Table “Property Sheet” in “Design View”?	70
B.6.4	How do I configure a Table “Property Sheet” in “Design View”?	71
B.7	What is a Query in “Design View”?	72
B.7.1	How do I unhide/show a Query’s “Property Sheet” in “Design View”?	73
B.7.2	How do I configure a Query’s “Property Sheet” in “Design View”?	74
B.8	What is a Form in “Design View”?	76
B.8.1	How do I unhide/show a Form’s “Property Sheet” in “Datasheet View” or	

“Design View”?	76
B.8.2 How do I configure a Form’s “Property Sheet” in “Datasheet View” or “Design View”?	77
B.9 What is a Query in “SQL View”?	79
B.10 What is the “Relationships” pane?	80
B.10.1 How do I open the “Relationships” pane?	80
B.10.2 How do I view my Relationships?	80
B.10.3 How do I view a Relationship’s properties?	82
B.10.4 How do I change the layout of the “Relationships” pane?	85
B.10.4.1 How do I select/unselect Table-boxes?	85
B.10.4.2 How do I unhide/hide Relationships?	86
B.10.4.3 How do I unhide/hide the “Add Tables” sub-pane?	86
B.10.4.4 How do I unhide or add Table-boxes?	87
B.10.4.5 How do I hide or delete a Table-box?	87
B.10.4.6 What is the difference between unhide/hide and add/delete a Table-box?	88
B.10.4.7 How do I move a Table-box?	88
B.10.4.8 How do I resize a Table-box?	89
B.10.4.9 How do I front-display a Table-box?	89
B.10.4.10 How do I save the “Relationships” pane layout?	89
B.10.5 How do I edit an existing Relationship?	89
B.10.6 How do I create a new Relationship?	90
B.10.7 How do I delete a Relationship?	90
B.10.8 How do I close the “Relationships” pane?	90
B.11 Can I use a drop-down/expression menu even if its icon is not shown?	91
PART C. CONCEPTS AND INTERNALS OF DATABASES	92
C.1 What are the main concepts of databases?	92
C.1.1 What is a database, a record and a field?	92
C.1.2 What is the difference between a database and a spreadsheet?	93
C.1.3 What is a database design?	94
C.1.4 What is an SQL Query?	94
C.1.5 What is the Structured Query Language (SQL)?	94
C.1.6 How is a database used?	94
C.2 What are objects, names, keywords, data types, constants, variables, operators, functions and expressions?	95
C.2.1 What is an object and a name?	95
C.2.2 What is a qualified field name?	95

C.2.3 What is a keyword?.....	96
C.2.4 What is a data type? .....	96
C.2.5 What is a constant? .....	96
C.2.6 What is a variable?.....	97
C.2.7 What is an operator? .....	97
C.2.8 What is a function? .....	98
C.2.9 What is an expression?.....	99
C.2.10 What is an SQL operation? .....	99
C.2.11 What is expression syntax?.....	100
C.3 What are fields, field value-lists, records and record-lists? .....	100
C.3.1 What are fields, field names and field types? .....	101
C.3.2 What is a record and a record type?.....	101
C.3.3 What is a record-list? .....	101
C.3.4 What is a field value-list? .....	102
C.4 What are database Tables?.....	103
C.5 What is a pointer? .....	103
C.6 What is a Null? .....	104
C.7 What are duplicate records and duplicate field values?.....	105
C.7.1 What are duplicate records?.....	105
C.7.2 What are duplicate field values? .....	106
C.8 What is indexing? .....	107
C.8.1 Why is indexing useful? .....	107
C.8.2 Can I create one index over a list of field names (called a composite index)? .....	110
C.8.3 What different types of indexes are there? .....	112
C.8.3.1 What are simple indexes and composite indexes?.....	112
C.8.3.2 What are indexes with duplicate values and without duplicate values? .....	112
C.8.3.3 What are indexes that ignore Nulls and not ignore Nulls? .....	113
C.8.3.4 What are indexes with Nulls and without Nulls? .....	113
C.8.3.5 What is an index without duplicate values and without Nulls? .....	113
C.8.4 What indexes can I configure in a given Table?.....	114
C.8.5 Why should I use indexing in my database?.....	116
C.8.5.1 Why indexing improves performance?.....	116
C.8.5.2 Why indexing prevents duplicate records in a Table?.....	116
C.8.5.3 Why indexing allows establishing Relationships between Tables?.....	117
C.9 How do I prevent duplicate field values and duplicate records? .....	117
C.9.1 How I do I prevent duplicate field values over a group of field name(s) in my	



Tables?.....	117
C.9.2 How do I prevent duplicate records in my Tables?.....	118
C.9.3 How do I prevent duplicate records in my Query's record-lists?.....	119
C.10 What are the Table Key(s) and how should I handle them?.....	120
C.10.1 What is the Primary Key of a Table?.....	120
C.10.2 What is a simple Key and a composite Key?.....	121
C.10.3 What are the candidate Keys of a Table?.....	122
C.11 What is a Relationship? .....	122
C.11.1 What is a Relationship with referential integrity? .....	125
C.11.2 What are "one-to-many" and "one-to-one" Relationships?.....	127
C.11.3 What is an indeterminate Relationship? .....	130
C.11.4 What is a many-to-many Relationship?.....	130
C.11.5 Why should I configure Relationships?.....	132
PART D. DESIGNING MY DATABASES WITH MS-ACCESS.....	133
How do I design my database? .....	133
How do I use my database? .....	134
How do I evolve my database design? .....	135
D.1 How do I create, close and open a database file? .....	135
D.1.1 How do I create a database file? .....	135
D.1.2 How do I close a database file? .....	136
D.1.3 How do I open an existing database file? .....	136
D.2 How do I carefully assign good names from the very beginning? .....	137
D.2.1 Why should I make field names short but clear?.....	138
D.2.2 Why should I use homogeneous field names across Tables and Queries?.	139
D.2.3 Why should I use structured names for Tables and Queries?.....	139
D.2.4 Why should I avoid certain elements in names? .....	140
D.2.4.1 Why should I avoid special characters in names? .....	140
D.2.4.2 Why should I avoid "<>" as a field name?.....	141
D.2.4.3 Why should I avoid non-English letters in names? .....	141
D.2.4.4 Why must I avoid control characters in names?.....	141
D.2.4.5 Why must I avoid using keywords as names?.....	142
D.2.4.6 Why should I avoid the suffix "_n" in Table names?.....	142
D.2.5 What are the MS-Access formal rules for identifiers? .....	143
D.2.6 When can I assign the same name to different objects and/or properties?.	144
D.3 How do I create and design a Table and its fields? .....	145
D.3.1 How do I create a Table? .....	145

D.3.2	How do I create my Table fields? .....	146
D.4	How do I configure a Table field data type and size? .....	146
D.4.1	What is the “Short Text” field type? .....	147
D.4.2	What is the “Long Text” field type? .....	148
D.4.3	What are the “Number” field types? .....	148
D.4.3.1	What is the “Number-Byte” field type-size? .....	148
D.4.3.2	What is the “Number-Integer” field type-size? .....	149
D.4.3.3	What is the “Number-Long Integer” field type-size? .....	149
D.4.3.4	What is the “Number-Single” field type-size? .....	149
D.4.3.5	What is the “Number-Double” field type-size? .....	150
D.4.3.6	What is the “Number-Replication ID” field type-size? .....	150
D.4.3.7	What is the “Number-Decimal” field type-size? .....	150
D.4.4	What is the “Currency” field type? .....	150
D.4.5	What is the “Date/Time” field type? .....	151
D.4.6	What is the “Date/Time extended” field type? .....	152
D.4.7	What is the “Yes/No” field type? .....	153
D.4.8	What is the “Calculated” field type? .....	153
D.4.9	What is the “Large Number” field type? .....	154
D.4.10	What is the “AutoNumber” field type? .....	155
D.4.11	What is the “OLE Object” field type? .....	155
D.4.12	What is the “Hyperlink” field type? .....	156
D.4.13	What is the “Attachment” field type? .....	156
D.4.14	What is the “Lookup Wizard...”? .....	157
D.5	How do I configure a Table field validation rule, indexing, and other properties? .....	157
D.5.1	What common Table field properties should I set? .....	158
D.5.1.1	What is the “Description” Table field property? .....	158
D.5.1.2	What is the “Format” Table field property? .....	158
D.5.1.3	What is the “Caption” Table field property? .....	159
D.5.1.4	What is the “Default Value” Table field property? .....	159
D.5.1.5	What is the field “Validation Rule” Table field property? .....	159
D.5.1.6	What is the field “Validation Text” Table field property? .....	161
D.5.1.7	What is the “Required” Table field property? .....	161
D.5.1.8	What is the “Indexed” Table field property? .....	162
D.5.1.9	What is the “Text Align” Table field property? .....	162

D.5.2	What data-type specific Table field properties should I set?.....	162
D.5.2.1	What is the “Field Size” Table field property?.....	163
D.5.2.2	What is the “Allow Zero Length” Table field property? .....	163
D.5.2.3	What is the “Decimal Places” Table field property? .....	163
D.5.2.4	What is the “Input Mask” Table field property? .....	163
D.5.2.5	What is the “Result Type” Table field property?.....	164
D.6	How do I configure the Primary Key field(s) of a Table?.....	164
D.6.1	How do I configure a Primary Key with only one field? .....	164
D.6.2	How do I configure a Primary Key with several fields? .....	165
D.6.3	How do I change the Primary Key of a Table? .....	165
D.6.4	What checks are done on Key fields when saving my Table design? .....	166
D.6.5	How do I identify the Primary Key field(s) of a Table?.....	166
D.6.6	Why should I define the Primary Key field(s) in Tables?.....	168
D.7	How do I add simple and/or composite index(es) to a Table? .....	168
D.7.1	How do I add simple indexes to a Table?.....	169
D.7.2	How do I add composite (and simple) indexes to a Table?.....	169
D.7.3	How do I reconfigure the indexes of a Table?.....	170
D.8	How do I configure the properties of a Table?.....	171
D.8.1	How do I configure a record validation rule?.....	172
D.8.2	How do I configure the record validation text?.....	173
D.8.3	How do I configure the “Subdatasheet Name” property? .....	173
D.9	How do I create and configure my Table Relationships?.....	174
D.9.1	How do I create a new Relationship? .....	175
D.9.1.1	How do I create a new Relationship in the most frequent case? .....	175
D.9.1.2	How do I create a new Relationship in less frequent cases? .....	176
D.9.2	How do I configure a Relationship? .....	177
D.9.3	What is the effect of “Enforce Referential Integrity”? .....	180
D.9.4	What is the effect of “Cascade Update Related Fields”?.....	180
D.9.5	What is the effect of “Cascade Delete Related Records”? .....	181
D.9.6	What Relationships should I configure?.....	181
D.10	How do I design MS-Access Forms? .....	182
D.10.1	What are MS-Access Forms? .....	182
D.10.2	How do I create an MS-Access Form?.....	182
D.10.3	How do I add/delete/update a field of a Form? .....	183
D.10.3.1	How do I add a field to a Form?.....	183
D.10.3.2	How do I delete a field from a Form? .....	183

D.10.3.3 How do I update a field of a Form?.....	184
D.10.4 How do I configure VBA action code associated to Form Events? .....	184
D.10.4.1 How do I configure a Form to have an associated VBA Module?.....	184
D.10.4.2 How do I enter/edit the VBA subroutine associated to a Form Event?.....	185
D.10.4.3 How does my VBA code modify field values in the Form record being edited? .....	186
D.10.4.4 Why should I write options in the VBA Modules of my Forms?.....	187
D.10.4.5 How do I close the VBA editor? .....	187
D.11 How do I configure the way to enter data (e.g., a drop-down menu) in a Table/Form field? .....	188
D.11.1 How do I configure a drop-down menu to enter data in a Table field?....	189
D.11.1.1 How do I configure a Table drop-down menu that takes its values from a Table/Query?.....	190
D.11.1.2 How do I configure a Table drop-down menu that takes its values from a Value List?.....	191
D.11.1.3 How do I configure a Table drop-down menu that takes its values from a Field List?.....	192
D.11.2 How do I configure a date-picker to enter a date in a Table field? .....	193
D.11.3 How do I configure type-in or checkbox to enter data in a Table/Form field? .....	193
D.11.4 How do I configure a drop-down menu or a date-picker to enter data in a Form field?.....	194
D.11.4.1 How do I configure a new drop-down menu directly in a Form field? .....	194
D.11.4.2 How do I configure an existing drop-down menu in a Form field? .....	196
D.11.4.3 How do I configure a new date-picker directly in a Form field?.....	197
D.11.5 What options can I set in a “Combo Box” Table/Form drop-down menu? .....	198
D.11.5.1 What are useful options in a “Combo Box” Table/Form drop-down menu? ..	198
D.11.5.2 What are the interactions between slave fields, “Limit to List” property and the field validation rule?.....	200
D.12 How do I use MS-Access Reports? .....	200
D.13 How do I share a database, having multiple concurrent users? .....	201
<b>PART E. ENTERING, MODIFYING AND DELETING MY DATABASE DATA .</b>	<b>202</b>
E.1 How do I edit one new or existing record? .....	203
E.1.1 How do I go to the new-record row? .....	204
E.1.2 How do I enter the record under edition?.....	205
E.2 How do I edit the field’s values of the record under edition?.....	206
E.2.1 How do I choose a field value? .....	207
E.2.2 How do I type-in a value in a field?.....	208

E.2.2.1	How do I type-in a value in a Yes/No field?.....	209
E.2.2.2	How do I type-in a value in a Short Text field?.....	209
E.2.2.3	How do I type-in a value in a Number, Currency or Large Number field? .....	209
E.2.2.4	How do I type-in a value in a Date/Time field? .....	209
E.2.2.5	How do I type-in a zero-time value in a Date/Time field? .....	210
E.2.2.6	How do I type-in a zero-date value in a Date/Time field?.....	212
E.2.2.7	What happens if the field “Format” property does not match the field type?...	212
E.2.3	Why is a value shown in a different way when the Table/Query/Form field’s value has been selected? .....	213
E.2.3.1	How is the formatting of a Number or Currency field changed when I select its field’s value? .....	213
E.2.3.2	How is the formatting of a Yes/No field changed when I select its field’s value? .....	214
E.2.3.3	How is the formatting of a Date/Time field changed when I select its field’s value? .....	214
E.3	How do I interactively delete existing records? .....	215
E.4	What is different about entering, modifying or deleting records from a Table or a Form?.....	215
E.5	How do I copy/cut and paste data between MS-Access and other applications? .....	216
E.5.1	How do I copy/cut data from an MS-Access Table, Form or Query result? .....	216
E.5.2	How do I paste data into MS-Access? .....	217
E.5.2.1	How do I paste into a field’s value?.....	217
E.5.2.2	How do I paste over a rectangle of fields?.....	217
E.5.2.3	How do I paste as new records?.....	218
E.5.2.4	What are the fields and field order when pasting?.....	219
E.5.3	How do I paste data copied from MS-Access into other applications? .....	219
E.5.3.1	How do I paste as plain text into Excel?.....	219
E.5.3.2	How do I paste as formatted data into Excel?.....	221
E.5.3.3	How do I paste as plain text into Word?.....	222
E.5.3.4	How do I paste as formatted text into Word? .....	222
E.5.3.5	What are the additional rows when pasting into Excel or Word?.....	223
E.5.3.6	How do I paste into a plain-text processor? .....	223
E.5.3.7	How do I paste back into MS-Access? .....	223
E.6	What checks are done when saving a field value, or entering or modifying a record? .....	223
E.6.1	What checks are done when saving a field value?.....	223

E.6.2	What checks are done when entering or modifying a record? .....	224
E.7	How do I bulk-change my Table/Form's data? .....	225
E.7.1	How do I bulk-modify my data using the "Find/Replace" tool? .....	225
E.7.2	How do I bulk-change my data with an external application?.....	227
E.7.3	How do I bulk-change my data using data-changing Queries? .....	227
E.7.4	Why should I backup my data before a bulk-change? .....	228
E.8	How do I upload my pre-existing data into my database? .....	228
E.9	Can I get inconsistent results out of my initial data in my database? .....	229
E.10	Why should I use "Compact and Repair Database"? .....	229
PART F.	WRITING SQL QUERIES TO USE MY DATABASE .....	231
F.1	What is the Structured Query Language (SQL)? .....	231
F.2	What version of SQL is this guide for?.....	232
F.3	Why should I write and run Queries?.....	232
F.4	What is an SQL operation and an SQL Query? .....	232
F.4.1	What is an SQL operation? .....	233
F.4.2	What is an SQL Query?.....	233
F.4.3	Why should I write my Queries in SQL?.....	233
F.4.4	What SQL Query editor should I use? .....	234
F.4.5	How do I create a Query?.....	234
F.4.6	How do I edit my SQL Queries without a plug-in Query editor? .....	235
F.5	How do I edit my SQL Queries with the plug-in "Access SQL Editor"? .....	235
F.5.1	How do I start with the plug-in "Access SQL Editor"? .....	236
F.5.1.1	How do I get and install the plug-in "Access SQL Editor"? .....	236
F.5.1.2	How do I open the plug-in "Access SQL Editor"?.....	236
F.5.1.3	What options should I set in the "Access SQL Editor"? .....	237
F.5.2	What is the user interface of the plug-in "Access SQL Editor"? .....	238
F.5.3	What are the toolbar commands in the "Access SQL Editor"? .....	239
F.5.4	How do I manage the Queries/Tables with the "Access SQL Editor"? .....	240
F.5.4.1	How do I open a Query/Table from the "Access SQL Editor"? .....	240
F.5.4.2	How do I select a Query/Table name in the "Access SQL Editor"? .....	241
F.5.4.3	How do I show a query pane in the "Access SQL Editor"?.....	242
F.5.4.4	How do I select a query pane in the "Access SQL Editor"? .....	242
F.5.4.5	How do I run a Query from the "Access SQL Editor"? .....	243
F.5.4.6	How do I save a Query from the "Access SQL Editor"? .....	244
F.5.4.7	How do I close a query pane in the "Access SQL Editor"? .....	244
F.5.4.8	How do I create a new Query from the "Access SQL Editor"? .....	245

F.5.4.9 How do I rename a Query from the “Access SQL Editor”?.....	245
F.5.4.10 How do I copy a Query from the “Access SQL Editor”?.....	246
F.5.4.11 How do I delete a Query from the “Access SQL Editor”?.....	246
F.5.5 How do I edit a Query in its query pane in the “Access SQL Editor”?.....	246
F.5.5.1 What are the specific editing functionalities of a query pane from the “Access SQL Editor”? .....	247
F.5.5.2 What are the conventional editing functionalities of a query pane from the “Access SQL Editor”? .....	248
F.5.6 How do I configure the layout of the “Access SQL Editor”? .....	249
F.5.6.1 What are the query panes in the “Access SQL Editor”? .....	250
F.5.6.2 What is the “Queries” pane in the “Access SQL Editor”? .....	250
F.5.6.3 What is the “Tables” pane in the “Access SQL Editor”? .....	251
F.5.6.4 What is the “Status” pane in the “Access SQL Editor”? .....	252
F.5.6.5 What is the “Results” pane in the “Access SQL Editor”? .....	252
F.5.6.6 What is a pane-space in the “Access SQL Editor”? .....	252
F.5.6.7 What is a sub-window in the “Access SQL Editor”? .....	253
F.5.6.8 What is a self-standing window of the “Access SQL Editor”? .....	255
F.5.6.9 How do I move an object pane, a sub-window or a self-standing window of the “Access SQL Editor”? .....	256
F.5.6.10 How do I convert between an object pane, a pane-space, a sub-window, or a self-standing window in the “Access SQL Editor”? .....	257
F.5.6.11 What can be a good layout for the “Access SQL Editor”?.....	258
F.6 What are the SQL operators I use to write my Queries?.....	259
F.6.1 What are the SQL consulting operations?.....	259
F.6.2 What are the SQL data-changing operators?.....	261
F.7 What is a Select operation and how do I write it?.....	261
F.7.1 What is the Select operator?.....	262
F.7.2 What are the three types of Select? .....	265
F.7.3 What is the dataflow of a Select? .....	267
F.7.4 What is the input record-list (“FROM” clause) of a Select?.....	270
F.7.5 What are the output fields (“SELECT” clause) of a Select? .....	271
F.7.6 What is the output record-list of a Select? .....	273
F.7.6.1 What are the output field values of a Select-no_aggreg?.....	273
F.7.6.2 What are the output field values of a Select-group_by_aggreg? .....	274
F.7.6.3 What are the output field values of a Select-total_aggreg?.....	276
F.7.6.4 What are the output records of a Select?.....	277
F.7.6.5 How many output records does a Select produce?.....	279

F.7.7	What is the “WHERE” clause of a Select? .....	280
F.7.8	What is the “DISTINCTROW” clause of a Select? .....	281
F.7.9	What is the “GROUP BY” clause of a Select-group_by_aggreg?.....	281
F.7.10	What is the “HAVING” clause of Select-group_by_aggreg or Select-total_aggreg?.....	282
F.7.11	What is the “DISTINCT” clause of a Select? .....	283
F.7.12	How do I use “ORDER BY” to order the output records of a Select?.....	284
F.7.12.1	How are the different data/field types ordered by the “ORDER BY” clause?.	287
F.7.13	What is the “TOP” clause of a Select? .....	288
F.7.14	How do I write a correct (syntax) Select? .....	289
F.7.15	How do I write a correct (syntax) Select-no_aggreg?.....	290
F.7.15.1	What is a syntax-example of a Select-no_aggreg?.....	291
F.7.15.2	What are the formal rules (syntax) to write a Select-no_aggreg?.....	292
F.7.16	How do I write a correct (syntax) Select-group_by_aggreg?.....	295
F.7.16.1	What is a syntax-example of a Select-group_by_aggreg? .....	296
F.7.16.2	What are the formal rules (syntax) to write a Select-group_by_aggreg?.....	298
F.7.17	How do I write a correct (syntax) Select-total_aggreg?.....	303
F.7.17.1	What is a syntax-example of a Select-total_aggreg? .....	304
F.7.17.2	What are the formal rules (syntax) to write a Select-total_aggreg?.....	305
F.7.18	What is an SQL aggregate function?.....	308
F.7.18.1	What is the “Count (*)” SQL aggregate function?.....	310
F.7.18.2	What is the “Count ()” SQL aggregate function?.....	310
F.7.18.3	What are the “First ()” and “Last ()” SQL aggregate functions?.....	310
F.7.18.4	What are the “Min ()” and “Max ()” SQL aggregate functions?.....	310
F.7.18.5	What are the “Sum ()” and “Avg ()” SQL aggregate functions?.....	311
F.7.18.6	What are the “StDev ()”, “StDevP ()”, “Var ()” and “VarP ()” SQL aggregate functions?.....	312
F.7.18.7	What is a summary and grouping of aggregate functions?.....	313
F.8	What is a Join operation and how do I write it? .....	314
F.8.1	What is “joining” two ordered records? .....	314
F.8.2	What are the Join operators? .....	316
F.8.2.1	How do the four Join operations compare?.....	319
F.8.2.2	Are Join operations commutative?.....	320
F.8.3	What are the input record-lists of a Join?.....	322
F.8.4	What are the output fields of a Join?.....	323
F.8.5	What is the output record-list of a “,” Cross-Join? .....	324



F.8.5.1	What are the output field values of a “,” Cross-Join?.....	324
F.8.5.2	What are the output records of a “,” Cross-Join? .....	324
F.8.5.3	How many output records does a “,” Cross-Join produce? .....	326
F.8.6	What is the output record-list of an “INNER JOIN”?.....	326
F.8.6.1	What are the output field values of an “INNER JOIN”? .....	327
F.8.6.2	What are the output records of an “INNER JOIN”? .....	327
F.8.6.3	How many output records does an “INNER JOIN” produce?.....	332
F.8.7	What is the output record-list of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)?.....	332
F.8.7.1	What are the output field values of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)? .....	333
F.8.7.2	What are the output records of an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”)? .....	333
F.8.7.3	How many output records does an Outer-Join (“LEFT JOIN” or “RIGHT JOIN”) produce? .....	336
F.8.8	What is the output record-list of a Full-Outer-Join? .....	337
F.8.8.1	What are the output field values of a Full-Outer-Join?.....	337
F.8.8.2	What are the output records of a Full-Outer-Join?.....	337
F.8.8.3	How many output records does a Full-Outer-Join produce?.....	341
F.8.9	What is the “ON” clause of “INNER JOIN” and Outer-Join (“LEFT JOIN” and “RIGHT JOIN”)?.....	341
F.8.10	How do I write a correct (syntax) Join? .....	343
F.8.10.1	What is a syntax-example of a Join?.....	343
F.8.10.2	What are the formal rules (syntax) to write a Join? .....	344
F.8.10.3	How do I nest Joins? .....	346
F.9	What is a Union operation and how do I write it?.....	347
F.9.1	What are the Union operators?.....	348
F.9.2	What are the input record-lists of a Union? .....	349
F.9.3	What are the output fields of a Union?.....	349
F.9.4	What is the output record-list of a Union? .....	351
F.9.4.1	What are the output field values of a Union?.....	351
F.9.4.2	What are the output records of a Union? .....	351
F.9.4.3	How many output records does a Union produce? .....	351
F.9.5	How do I write a correct (syntax) Union?.....	352
F.9.5.1	What is a syntax-example of a Union? .....	352
F.9.5.2	What are the formal rules (syntax) to write a Union?.....	353
F.9.5.3	How do I nest Unions?.....	353

F.9.6 Why do I find misleading the names of the Union operators?.....	354
F.10 What is a Transform operation and how do I write it?.....	354
F.10.1 What is the Transform operator?.....	356
F.10.2 What is the input record-list (“FROM” clause) of a Transform? .....	359
F.10.3 What are the output fields of a Transform?.....	359
F.10.3.1 What is the number of output fields of a Transform? .....	359
F.10.3.2 What are the output field names of a Transform?.....	360
F.10.3.3 What is the output field order of a Transform?.....	362
F.10.3.4 What are the output data/field types of a Transform?.....	363
F.10.4 What is the output record-list of a Transform? .....	364
F.10.4.1 What are the output field values of a Transform?.....	364
F.10.4.2 What are the output records of a Transform?.....	367
F.10.4.3 How many output records does a Transform produce?.....	368
F.10.5 Can I see an example of a Transform operation?.....	368
F.10.6 What is the “TRANSFORM” clause of a Transform?.....	371
F.10.7 What is the “SELECT” clause of a Transform?.....	371
F.10.8 What is the “ORDER BY” clause of a Transform? .....	371
F.10.9 What is the “WHERE” clause of a Transform? .....	371
F.10.10 What is the “GROUP BY” clause of a Transform? .....	372
F.10.11 What is the “PIVOT” clause of a Transform? .....	372
F.10.12 What is the “IN” clause of a Transform?.....	373
F.10.13 How do the clauses from Transform and Select compare? .....	374
F.10.14 How do I write a correct (syntax) Transform?.....	374
F.10.14.1 What is a syntax-example of a Transform?.....	375
F.10.14.2 What are the formal rules (syntax) to write a Transform? .....	377
F.10.14.3 Can I nest Transform operations? .....	382
F.11 What are the SQL clauses, their expression’s elements and color codes? .....	382
F.11.1 Given an SQL clause, what are its expression’s elements? .....	382
F.11.2 What are the SQL color codes used in this Guide?.....	385
F.12 How do I add parameters (type-in variables) to my Queries?.....	386
F.13 How do I write a Query that changes my Table data? .....	388
F.13.1 What is a Delete operation and how do I write it?.....	389
F.13.2 What is an Insert operation and how do I write it? .....	389
F.13.2.1 How do I write a Query that inserts many Table records?.....	390
F.13.2.2 How do I write a Query that inserts only one Table record? .....	391
F.13.2.3 What are the common characteristics to all Insert operations?.....	392

F.13.2.4 What is the summary of Insert operations?.....	395
F.13.3 What is an Update operation and how do I write it?.....	396
F.13.4 Can I write a VBA function that deletes, inserts or updates Table records? .....	396
F.13.5 When should I use SQL operations that change records from my Tables? .....	396
F.14 How do I write and debug my SQL Queries? .....	396
PART G. WRITING EXPRESSIONS .....	398
G.1 What are the main differences between the three expression scopes? .....	398
G.1.1 How available field names depend on expression scopes? .....	399
G.1.2 How writing field names depends on expression scopes?.....	400
G.1.3 How data types depend on expression scopes? .....	400
G.1.4 How constants depend on expression scopes?.....	400
G.1.5 How value operators depend on expression scopes?.....	400
G.1.6 How non-aggregate built-in functions depend on expression scopes?.....	401
G.1.7 How domain aggregate functions depend on expression scopes?.....	402
G.1.8 How SQL aggregate functions depend on expression scopes? .....	402
G.1.9 How user-defined functions depend on the expression scopes?.....	402
G.1.10 How using SQL operations (“Subqueries”) depends on the expression scopes?.....	402
G.2 How do I manage VBA data types and Table field types-sizes?.....	403
G.2.1 What VBA data types vs. Table field types-sizes are equivalent? .....	403
G.2.2 What VBA data types vs. Table field types-sizes are not equivalent? .....	404
G.2.3 What is the result of combining different data/field types in expressions? .....	405
G.2.4 How are data/field types grouped for easier reference? .....	406
G.2.5 How do I force a value to belong to a specific data type?.....	408
G.2.6 Why should I force a value into a specific data type? .....	409
G.3 What is the data type returned by an expression?.....	410
G.3.1 What is the data type of a constant? .....	410
G.3.2 What is the data type returned by a non-arithmetic operator?.....	410
G.3.3 What is the data type returned by an arithmetic operator? .....	411
G.3.4 What is the data type returned by a function? .....	411
G.4 How do I write a constant? .....	411
G.4.1 How do I write <i>Boolean</i> constants?.....	412
G.4.2 How do I write <i>String</i> constants?.....	412
G.4.3 How do I write <i>Byte</i> , <i>Integer</i> , <i>Long Integer</i> , <i>LongLong</i> , <i>Currency</i> , <i>Single</i> and <i>Double</i> constants?.....	413

G.4.4	How do I write <i>Date</i> constants? .....	413
G.4.5	What are the restrictions when writing constants in VBA expressions?....	413
G.5	How do I use value operators in an expression?.....	414
G.5.1	What are the Arithmetic operators?.....	415
G.5.2	What are the Text string operators?.....	416
G.5.3	What are the Comparison operators?.....	417
G.5.4	What are the Pattern operators?.....	418
G.5.5	What are the Logical ( <i>Boolean</i> ) operators?.....	419
G.5.6	What are the “IS NULL” and “IS NOT NULL” Miscellaneous operators? .....	420
G.5.7	What are the “IN” and “NOT IN” Miscellaneous operators? .....	421
G.5.8	What are the “BETWEEN AND” and “NOT BETWEEN AND” Miscellaneous operators?.....	422
G.6	How do I use functions in an expression? .....	423
G.6.1	How do I use non-aggregate built-in functions in an expression?.....	424
G.6.2	How do I use domain aggregate functions in an expression?.....	426
G.6.3	How do I use SQL aggregate functions in an expression? .....	428
G.6.4	How do I use user-defined VBA functions in an expression?.....	429
G.7	What is the evaluation order of an expression? .....	429
G.8	How do I use an SQL operation in an expression?.....	430
G.8.1	How do I use a Subquery in an SQL expression? .....	431
G.8.2	What are the “EXISTS”, “ANY” and “ALL” Subquery operators?.....	432
G.8.2.1	What is the “EXISTS” Subquery operator?.....	432
G.8.2.2	What is the “ANY” Subquery operator?.....	433
G.8.2.3	What is the “ALL” Subquery operator?.....	433
G.8.3	What is a correlated Subquery? .....	434
G.8.4	What is an uncorrelated Subquery? .....	434
G.8.5	How do I use an SQL operation in a VBA variable assignment? .....	434
G.9	How are numeric-like values internally represented and processed?.....	435
G.9.1	How are numeric-like values internally represented? .....	435
G.9.2	What is decimal/binary conversion?.....	435
G.9.3	What are decimal/binary conversion rounding errors?.....	436
G.9.4	What is the cause of decimal/binary conversion rounding errors?.....	437
PART H.	CUSTOMIZING THE APPEARANCE OF A QUERY/TABLE/FORM IN “DATASHEET VIEW”.....	439
H.1	How do I change the column width, or freeze/unfreeze the columns in a Table/Query/Form?.....	439

H.1.1	How do I change the column width in a Table/Query/Form? .....	439
H.1.2	How do I freeze/unfreeze the columns in a Table/Query/Form? .....	440
H.1.2.1	How do I freeze the columns in a Table/Query/Form? .....	441
H.1.2.2	How do I unfreeze all the columns in a Table/Query/Form? .....	441
H.2	How do I change row height, hide rows or change row order in a Table/Query/Form?.....	442
H.2.1	How do I change the height of all the rows in a Table/Query/Form? .....	442
H.2.2	How do I hide rows in a Table/Query/Form? .....	443
H.2.3	How do I change the sorting of rows in a Table/Query/Form? .....	443
H.3	How do I change the order of columns that I see in a Table/Query/Form?.....	445
H.3.1	How do I change the order of columns in a Table/Query/Form? .....	445
H.3.2	How is it related a Table's column order and its field order in "Design View"? .....	446
H.3.3	How is it related a Query's column order and its field order in "Design View" and "SQL View"? .....	446
H.3.4	How is it related a Forms' column order and shown/hidden columns and the ones of its associated Table? .....	447
H.4	How do I hide/unhide columns in a Table/Query/Form? .....	447
H.4.1	How do I hide the columns that I see in a Table/Query/Form?.....	448
H.4.2	How do I unhide columns in a Table/Query/Form? .....	448
H.4.3	How are related a Table/Query's shown/hidden columns and its fields in other views? .....	449
H.4.4	How is it related a Forms' column order and shown/hidden columns and the ones of its associated Table? .....	449
H.5	How do I change the column headings in a Table/Query/Form? .....	450
H.5.1	How do I change a Table's column headings? .....	450
H.5.2	How do I change a Query's column headings? .....	451
H.5.3	How do I change a Form's column headings?.....	451
H.5.4	How are related a Form's column headings and the ones of its associated Table? .....	452
H.6	How do I configure the formatting of column values in a Table/Query/Form? .....	452
H.6.1	How do I configure predefined column formatting in a Table/Query/Form? .....	453
H.6.1.1	What "Format" options are available in Short Text and Long Text fields? .....	453
H.6.1.2	What "Format" options are available in Yes/No fields?.....	453
H.6.1.3	What "Format" options are available in Number, Large Number, Currency and Date/Time fields? .....	454
H.6.1.4	What "Format" options are available in Calculated fields? .....	456

H.6.2	How do I configure custom column formatting in a Table/Query/Form?..	457
H.6.2.1	How do I configure custom formatting for all field types? .....	457
H.6.2.2	How do I configure custom formatting for Short Text fields? .....	458
H.6.2.3	How do I configure custom formatting for Number, Large Number and Currency fields? .....	459
H.6.2.4	How do I configure custom formatting for Date/Time field type? .....	460
H.6.3	How do I find the column formatting properties in a Table/Query/Form?	461
H.6.3.1	How do I find a Table’s column formatting properties? .....	462
H.6.3.2	How do I find a Query’s column formatting properties? .....	462
H.6.3.3	How do I find a Form’s column formatting properties?.....	463
H.7	How do I configure the column text alignment in a Table/Query/Form? .....	464
H.7.1	How do I configure a Table/Form’s column text alignment?.....	464
H.7.2	How do I configure a Query’s column text alignment? .....	465
H.8	How do I show aggregate values (e.g., totals) in a Table/Query/Form? .....	465
H.8.1.1	How do I show aggregate values in Short Text and Yes/No fields?.....	466
H.8.1.2	How do I show aggregate values in Date/Time field? .....	466
H.8.1.3	How do I show aggregate values in Number, Large Number and Currency fields? .....	467
H.9	How do I configure colors, fonts and other features of a Table/Query/Form?..	467
PART I. EVOLVING MY DATABASE DESIGN.....		469
I.1	Why would I want to improve/modify my database design?.....	469
I.2	Why should I be so careful with any change to the database design? .....	469
I.2.1	What are database element dependencies?.....	470
I.2.2	What are side effects? .....	471
I.3	How do I find all the dependent objects on a given database object?.....	472
I.4	What are the side effects of modifying my Table fields? .....	472
I.4.1	What are the side effects of adding a field to a Table? .....	472
I.4.2	What are the side effects of deleting a Table field? .....	474
I.4.3	What are the side effects of changing the name of a Table field? .....	476
I.4.4	What are the side effects of modifying the properties of a Table field? .....	477
I.4.4.1	What are the side effects of changing the “Field Type” and/or “Field Size” properties of a Table field?.....	477
I.4.4.2	What are the side effects of changing the “Required”, “Allow zero length”, “Validation rule” or “Indexing” properties of a Table field?.....	481
I.4.4.3	What are the side effects of changing a drop-down menu of a Table field?.....	481
I.4.5	What are the side effects of changing the order of Table fields?.....	481
I.5	What are the side effects of modifying my Tables?.....	482

I.5.1	What are the side effects of adding a new Table?.....	483
I.5.2	What are the side effects of deleting a Table? .....	483
I.5.3	What are the side effects of changing the name of a Table?.....	484
I.5.4	What are the side effects of modifying a Table field? .....	484
I.5.5	What are the side effects of modifying the order of fields in the Table? .....	484
I.5.6	What are the side effects of modifying the indexes or the Key fields of a Table? .....	484
I.5.6.1	What are the side effects of changing an index with duplicate values? .....	485
I.5.6.2	What are the side effects of adding an index without duplicate values?.....	485
I.5.6.3	What are the side effects of changing an index without duplicate values?.....	485
I.5.6.4	What are the side effects of adding or changing the Key fields of a Table?.....	486
I.5.7	What are the side effects of modifying the record validation rule of a Table? .....	486
I.6	What are the side effects of modifying my Queries? .....	486
I.6.1	What are the side effects of adding a new Query? .....	487
I.6.2	What are the side effects of deleting a Query?.....	487
I.6.3	What are the side effects of modifying the functionality of a Query? .....	487
I.6.4	What are the side effects of changing the name of a Query? .....	488
I.6.5	What are the side effects of adding a new field to a Query?.....	489
I.6.6	What are the side effects of deleting a Query field? .....	490
I.6.7	What are the side effects of changing the name of a Query field?.....	491
I.6.8	What are the side effects of changing the data type of a Query field?.....	491
I.6.9	What are the side effects of modifying the order of fields in a Query? .....	492
I.7	What are the side effects of modifying my user-defined VBA functions? .....	493
I.8	What are the side effects of modifying my Relationships, Forms and/or Reports? .....	494
PART J.	DEBUGGING MY SQL QUERIES.....	495
J.1	How do I fix an error/crash in a test-and-proven Query? .....	495
J.2	How do I fix an error/crash in a non-test-and-proven Query?.....	497
J.3	How do I debug by commenting/uncommenting? .....	497
J.4	How do I debug in progressive steps? .....	498
J.5	How do debug inside out?.....	499
J.6	How do I debug my same-level code linearly?.....	499
J.7	How do I debug the current uncommented SQL operation at each step?.....	500
J.7.1	How do I debug the current uncommented SQL operation so it can be saved? .....	501
J.7.2	How do I debug the current uncommented SQL operation so it runs?.....	504

J.7.3	How do I debug the current uncommented SQL operation so it does not produce defective results? .....	506
J.7.4	How do I debug the current uncommented SQL operation's functionality? .....	507
J.7.5	How do I check the results of the current uncommented SQL operation? ..	509
J.8	How do I fix a syntax error that prevents saving a Query?.....	510
J.9	How do I fix a crash from a syntax error? .....	511
J.10	How do I fix a crash from a run-time error? .....	517
J.10.1	How do I fix the crash " <i>Divide by zero.</i> "? .....	517
J.10.2	How do I fix the crash " <i>Overflow.</i> "? .....	518
J.10.3	How do I fix the crash " <i>Data type mismatch in criteria expression.</i> "? .....	519
J.10.4	How do I fix the crash " <i>Data type mismatch.</i> "? .....	520
J.10.5	How do I fix the crash " <i>Invalid Procedure Call or Argument.</i> "? .....	521
J.10.6	How do I fix the crash " <i>Invalid use of Null.</i> "? .....	521
J.10.7	How do I fix the crash " <i>The expression cannot be used in a calculated column.</i> "? .....	521
J.10.8	How do I fix the crash " <i>Cannot have ... in aggregate argument.</i> "? .....	522
J.10.9	How do I fix the crash " <i>Could not find field...</i> "? .....	522
J.10.10	How do I fix the crash " <i>The expression cannot be used in a Calculated column</i> "? .....	522
J.10.11	How do I fix the crash " <i>Expression is too complex.</i> "? .....	523
J.10.12	How do I fix the crash " <i>Query is too complex.</i> "? .....	523
J.10.13	How do I fix the crash " <i>Cannot open any more databases.</i> "? .....	523
J.10.14	How do I fix a crash from my user-defined VBA functions? .....	523
J.11	How do I fix defective Query results? .....	524
J.11.1	How do I fix a Query showing a wrong numeric-like value? .....	525
J.11.2	How do I fix a Query showing a wrong Short Text value? .....	525
J.11.3	How do I fix a Query erroneously considering two different values as equal? .....	525
J.11.4	How do I fix a Query erroneously considering two equal values as different? .....	526
J.11.5	How do I fix a Query erroneously ordering records? .....	527
J.11.6	How do I fix a Query requesting a non-declared parameter? .....	527
J.11.7	How do I fix a Query requesting a non-existing parameter? .....	528
J.11.8	How do I fix a Query requesting the same parameter twice (or more times)? .....	528
J.11.9	How do I fix a Query showing "#####" in a field? .....	529
J.11.10	How do I fix a Query producing "#Num!", "#Div/0!", "#Error", "#Type!" or	



“#Func!” in a field?.....	529
J.11.11 How do I fix a Query producing “#Invalid” or “#Deleted” in a field?....	530
J.11.12 How do I fix a Query showing a black square in a field?.....	531
J.11.13 How do I fix a Query producing “#Name?” in a field? .....	531
J.11.14 How do I fix a Query showing a blank field that should not be blank?...	531
J.11.15 How do I fix a Query producing defective values? .....	532
J.11.16 How do I fix a Query’s defective “ON” expression? .....	533
J.11.17 How do I fix a Transform Query producing wrong field names?.....	533
J.11.18 How do I fix a Query that stalls/freezes?.....	534
J.11.19 How do I fix a Query making arithmetic errors? .....	534
J.11.20 How do I fix a Query making rounding errors?.....	535
J.11.21 How do I fix a Query that I cannot open in “Design View”? .....	539
J.11.22 How do I fix my VBA functions comparing text strings case sensitive?	539
J.12 What do I do when I just cannot fix a Query? .....	539
J.13 Why should I always compare the results of an existing Query? .....	540
J.13.1 What is a record-list comparator tool?.....	541
J.14 What Null-related bugs can I get? .....	542
J.14.1 What effect does Null cause in Query results? .....	542
J.14.2 What effects does Null cause in Arithmetic, Comparison and Pattern value operators?.....	542
J.14.3 What effects does Null cause in Logical value operators? .....	542
J.14.4 What effects does Null cause in Text string value operators? .....	543
J.14.5 What effects does Null cause in the “IN” and “NOT IN” Miscellaneous value operators?.....	543
J.14.6 What effects does Null cause in SQL operators that remove duplicate records? .....	543
J.14.7 What effect does Null cause in Union SQL operators? .....	543
J.14.8 What effects does Null cause in aggregate functions?.....	544
J.14.9 What effects does Null cause as an argument of a VBA function? .....	544
J.15 What exception-value bugs can I get? .....	545
J.15.1 What is an exception-value? .....	545
J.15.2 What operations produce number-overflow “#Num!”?.....	546
J.15.3 How do I prevent exception-values?.....	547
J.15.4 What is the effect of an exception-value in value operators? .....	548
J.15.5 What is the effect of an exception-value in function arguments?.....	548
J.15.6 What is the effect of an exception-value in expressions? .....	548
J.15.7 When is an exception-value crashing the Query?.....	548

J.16	What data type bugs can I get? .....	549
J.16.1	What data type bugs can I get with constants? .....	549
J.16.2	What data type bugs can I get with value operators?.....	550
J.16.2.1	What data type problems can I get with Arithmetic operators?.....	550
J.16.2.2	What data type problems can I get with Text string operators?.....	551
J.16.2.3	What data type problems can I get with Comparison operators? .....	551
J.16.2.4	What data type problems can I get with Pattern operators?.....	552
J.16.2.5	What data type problems can I get with Logical operators?.....	552
J.16.2.6	What data type problems can I get with Miscellaneous operators?.....	552
J.16.3	What data type bugs can I get with the Union operator? .....	552
J.16.4	What data type bugs can I get with aggregate functions?.....	553
J.16.5	What data type bugs can I get with VBA functions?.....	553
PART K.	USEFUL DESIGN ADVICE .....	555
K.1	What are good practices in my Table design? .....	555
K.1.1	Why should I write field descriptions in my Table fields?.....	555
K.1.2	Why should I add validation rules to my Tables? .....	556
K.1.2.1	Why should I add field validation rules to my Table fields?.....	556
K.1.2.2	Why should I add record validation rules to my Tables?.....	557
K.1.3	Why should I prevent Nulls in my Table fields?.....	557
K.1.4	Why should I add a “Comments” field to my Tables?.....	558
K.1.5	Why should I add at least one Date/Time field to my Tables? .....	558
K.1.6	How to prevent errors in Short Text fields?.....	560
K.1.7	Why should I configure drop-down menus to enter data?.....	561
K.1.8	What are good practices in configuring my drop-down menus?.....	561
K.1.8.1	What “Row Source Type” and “Row Source” should I use in my drop-down menus?.....	562
K.1.8.2	Should I create a Relationship from an auxiliary Table used in drop-down menus?.....	563
K.1.8.3	Should I configure “Limit to List=Yes” in my drop-down menus? .....	564
K.1.9	How do I configure a drop-down menu in a Date/Time field? .....	565
K.2	What are other good practices in my database design? .....	566
K.2.1	Why should I hide unfrequently used objects?.....	566
K.2.2	Why should I add a “T_Numbers” Table with integer numbers?.....	567
K.2.3	What are the interactions between Nulls, duplicates, indexing, and Key field(s)?.....	567
K.2.4	What are the interactions between Relationships and “PrimaryKey”, “Required” and Table indexes? .....	569

K.2.5	What are the actual fields and the actual field order in a Table/Query/Form?	569
K.2.6	What is the difference between a Calculated field and automatically introducing a value using a Form?.....	570
K.2.7	How should I back-up MS-Access database files?.....	571
K.2.8	How do I store a list of values, instead of a single value, in a Table field?	571
K.2.9	Why and how should I maximize Table data correctness and coherence?.	572
K.2.10	How do I restrict the values introduced in my Table fields?.....	572
K.3	How do I structure and optimize a distributed database?.....	577
K.3.1	How do I organize my database files?.....	577
K.3.2	How do I create user-specific views of my shared database? .....	579
K.3.3	What are frontend files, backend files and source files? .....	580
K.3.4	What are local Tables, linked Tables and source Tables?.....	580
K.3.5	How do I get a split database? .....	582
K.3.6	How do I create a linked Table? .....	583
K.3.7	How do I refresh a Table link? .....	584
K.3.8	How do I convert a linked Table to a local Table?.....	584
K.3.9	How do I view a Table link? .....	585
K.3.10	How do I view and manage Table links? .....	585
K.3.11	How do I delete a linked Table? .....	588
K.3.12	How do I manage Relationships with linked Tables? .....	588
K.3.13	Can I change the file path or name of a source file? .....	589
K.3.14	What are the options to lock records in a shared database? .....	589
K.3.15	What is the delay of network access to a database? .....	590
K.3.16	Why should I make temporal Tables local? .....	591
K.4	What Query design principles should I follow?.....	591
K.4.1	How do I write my SQL Queries?.....	592
K.4.2	Why should I incrementally run my SQL code while I write a Query? .....	592
K.4.3	How do I write readable (maintainable) SQL Queries? .....	593
K.4.4	Why should I disable changing Table data from Query results?.....	599
K.4.5	Why should I qualify all the outermost output field names of my Queries?	600
K.4.6	When should I remove duplicate records? .....	600
K.4.7	Why should I enclose the outermost Union operation in a Select operation?	601
K.4.8	Why should I restrict the usage of “INNER JOIN”?.....	602
K.4.9	Why should I avoid using “SELECT *”? .....	602

K.4.10 Why should I avoid using the same name of an input field name for an output expression? .....	603
K.4.11 When are “SELECT” expressions evaluated along Query processing? ...	603
K.5 Why and how should I carefully handle Nulls in my Queries? .....	604
K.5.1 What is a Null? .....	604
K.5.2 What problems can Null produce? .....	604
K.5.3 How is a Null produced? .....	604
K.5.4 How do I handle Nulls in my Queries? .....	605
K.5.5 Where do I handle Nulls in my Queries? .....	607
K.5.6 Why is MS-Access not handling Null fields as I indicated? .....	607
K.5.6.1 How are Null fields created in “LEFT JOIN” and “RIGHT JOIN”?.....	608
K.5.6.2 Why should I handle Null fields in all the innermost SQL operations? .....	609
K.6 What are some useful models of SQL code?.....	613
K.6.1 How do I generate/create record-lists? .....	614
K.6.2 How do I replicate record-lists?.....	615
K.6.3 How do I produce totals in addition to individual results?.....	616
K.6.4 How do I convert rows into columns?.....	618
K.6.5 How do I convert columns into rows?.....	621
K.6.6 How do I use a Transform operation to show a cross table?.....	622
K.6.7 How do I produce the non-matching records of a Join operation?.....	624
K.6.8 How do I write a Full-Outer-Join?.....	627
K.6.9 How do I “merge” two record-lists?.....	629
K.6.10 How do I produce a weighted average?.....	632
K.6.11 How do I design a data check Query? .....	634
K.6.12 How do I get the exact record ordering I want? .....	636
K.6.12.1 How do I order over several expressions?.....	636
K.6.12.2 How do I order over only one expression?.....	636
K.6.12.3 How do I order over user-defined functions?.....	637
K.6.12.4 How do I order over “GROUP BY” expressions? .....	637
K.6.12.5 How do the four ordering approaches compare among themselves? .....	637
K.6.13 How do I use <i>String</i> fields to display different data types in the same Query column?.....	638
K.6.14 What are useful tricks with SQL aggregate functions? .....	639
K.6.14.1 How do I use the “AllNull ()” SQL aggregate function?.....	640
K.6.14.2 How do I use the “AllEqual ()” SQL aggregate function? .....	640
K.6.14.3 How do I use the “Or ()” or “And ()” SQL aggregate functions? .....	641
K.7 Why and how do I design a fast database and fast Queries? .....	642

K.7.1 How do I design a fast database?.....	643
K.7.2 How do I design faster Select operations over Queries? .....	644
K.7.2.1 Why should I use “DISTINCT”, “UNION” and “ORDER BY” only if needed? .....	644
K.7.2.2 Why should I use the most restrictive “WHERE” and “HAVING” expressions in my Select operations?.....	645
K.7.2.3 Why should I mainly use comparison and logical operators in my “WHERE” and “HAVING” expressions? .....	646
K.7.3 How do I design faster Select operations over Tables?.....	647
K.7.3.1 Why should I avoid bringing unnecessary data from Tables?.....	647
K.7.3.2 Why should I consider using uncorrelated Subqueries and/or domain aggregate functions in “WHERE” expressions over Tables? .....	649
K.7.4 How do I design faster Union operations? .....	650
K.7.5 How do I design faster Join operations?.....	650
K.7.5.1 Why should I always restrict in inner Selects?.....	650
K.7.5.2 Why should I mainly use comparison and logical operators in my “ON” expressions? .....	651
K.7.6 How do I design a faster Select-group_by when I have bound values in all my records?.....	652
K.8 Why should I avoid using Decimal data types?.....	653
K.8.1 What are the Decimal data types? .....	653
K.8.2 Why should I avoid using the Number-Decimal Table field type? .....	653
K.8.3 Why should I avoid using the <i>Variant-Decimal</i> VBA data type? .....	654
K.8.4 How do Decimal data types work? .....	655
K.9 How do I write my user-defined VBA functions and database Subroutines? ...	655
K.9.1 How do I create VBA modules and open the VBA editor?.....	656
K.9.2 How do I write a VBA function that inserts/updates/deletes Table’s records? .....	658
K.9.3 How do I write a VBA function that reads database records?.....	659
K.9.4 How do I write a VBA function that requests a parameter to the user? .....	660
K.9.5 How do I test my user-defined VBA functions? .....	660
K.9.6 How do I write VBA Subroutines associated to Form Events?.....	662
K.9.7 How do I close the VBA editor? .....	662
K.10 What elements/concepts are explained in various places? .....	663
K.10.1 Where are names and identifiers explained? .....	663
K.10.2 Where are drop-down menus explained? .....	663
K.10.3 Where are duplicates explained? .....	664
K.10.4 Where are indexes explained? .....	664

K.10.5	Where are Key fields explained?.....	664
K.10.6	Where are Relationships explained?.....	665
K.10.7	Where is Query results ordering explained?.....	665
K.10.8	Where are data types explained? .....	665
K.10.9	Where are zero-length and invisible strings explained? .....	666
K.10.10	Where are exception-values explained? .....	666
K.10.11	Where are aggregate functions explained? .....	666
K.10.12	Where is remote database access explained?.....	667
K.10.13	Where is VBA code explained?.....	667
K.10.14	Where are Nulls explained?.....	667
PART L	FIXING DATABASE ERRORS .....	669
L.1	Why can I get an error/crash in a test-and-proven database?.....	669
L.2	How do I fix errors with my Table/Form design?.....	670
L.2.1	How do I fix data integrity errors when saving my Table design? .....	670
L.2.1.1	How do I fix “Nulls in Required field” when saving my Table design?.....	673
L.2.1.2	How do I fix “field validation rule violated” when saving my Table design?..	673
L.2.1.3	How do I fix “record validation rule violated” when saving my Table design?673	
L.2.1.4	How do I fix field Type-size change errors when saving my Table design? ....	674
L.2.1.5	How do I fix that I cancelled the Table data integrity check? .....	674
L.2.2	How do I fix orphan Calculated fields when saving my Table design?.....	675
L.2.3	How do I fix “Calculated column cannot be saved...” when saving my Table design? .....	675
L.2.4	How do I fix “Could not find field...” when writing the expression of a Calculated field? .....	675
L.2.5	How do I fix “Could not find field...” when saving my Table design?.....	675
L.2.6	How do I fix “Key/Index with duplicate values” when saving my Table design? .....	676
L.2.7	How do I fix “...primary key cannot contain a Null...” when saving my Table design? .....	677
L.2.8	How do I fix “You can’t change the primary key” when saving my Table design? .....	677
L.2.9	How do I fix “Cannot delete this index...” when saving my Table design? 678	
L.2.10	How do I fix “There is no primary key...” when saving my Table design?678	
L.3	How do I fix errors in my Relationship configuration? .....	679
L.3.1	How do I fix “different number of fields” Relationship error?.....	679
L.3.2	How do I fix “different field types” Relationship error? .....	680
L.3.3	How do I fix “different field sizes” Relationship error? .....	680

L.3.4	How do I fix “violates referential integrity” Relationship error? .....	681
L.3.5	How do I fix “missing index” Relationship error?.....	681
L.3.6	How do I fix “...could not lock table...” Relationship error? .....	682
L.3.7	How do I fix a removed Relationship when I create a new one?.....	682
L.3.8	How do I fix showing a wrong Relationship type?.....	682
L.3.9	How do I fix “reversed Relationship Tables and fields” error?.....	683
L.3.10	How do I fix that I cannot configure a Relationship between a Table and itself?.....	684
L.3.11	How do I fix a malfunctioning Table slave record?.....	684
L.3.12	How do I fix a malfunctioning Relationship? .....	684
L.4	How do I fix errors when entering, modifying or deleting records?.....	685
L.4.1	How do I fix various field value errors? .....	685
L.4.1.1	How do I fix that I cannot type-in a text string in full? .....	685
L.4.1.2	How do I fix that I cannot paste into a field’s value? .....	685
L.4.1.3	How do I fix a changed numeric value? .....	686
L.4.1.4	How do I fix a changed Short Text value that I typed-in? .....	687
L.4.1.5	How do I fix a Table Short Text field configured as “Required=Yes” that accepts a Null? .....	687
L.4.1.6	How do I fix typing-in invisible characters into a Short Text Table field?.....	687
L.4.2	How do I fix error messages when saving a field value?.....	688
L.4.2.1	How do I fix an invalid value”?.....	688
L.4.2.2	How do I fix trying to save Null in a “Required” field?.....	689
L.4.2.3	How do I fix violating “Allow Zero Length=No”? .....	689
L.4.2.4	How do I fix that I cannot paste a text string in full?.....	689
L.4.2.5	How do I fix a value rejected because it is not in the list?.....	690
L.4.2.6	How do I fix a value violating a field validation rule? .....	690
L.4.3	How do fix error messages when entering a record? .....	690
L.4.3.1	How do I fix a record violating a record validation rule?.....	690
L.4.3.2	How do I fix erroneous records because of duplicate values?.....	691
L.4.3.3	How do I fix a slave record without a master record? .....	691
L.4.3.4	How do I fix an erroneous slave record? .....	692
L.4.4	How do I fix errors when pasting records into a Table/Form? .....	692
L.4.4.1	What pasting errors can I get from invisible characters in Text fields?.....	694
L.4.4.2	What pasting errors can I get from data copied from Excel?.....	695
L.4.4.3	What pasting errors can I get from Windows’ cut/paste buffer? .....	696
L.4.5	How do I fix a record I cannot delete?.....	696
L.5	How do I fix errors in Table/Form data? .....	696

L.5.1 How do I fix a Table/Form showing “#####” in a field? .....	697
L.5.2 How do I fix a Table/Form showing “#Num!”, “#Div/0!”, “#Type!” or “#Func!” in a field?.....	698
L.5.3 How do I fix a Table/Form showing “#Invalid” in a field?.....	698
L.5.4 How do I fix a Table/Form showing “#Deleted” in a field?.....	699
L.5.5 How do I fix a Form showing “#Name?” in a field? .....	699
L.5.6 How do I fix a Table/Form showing a black square in a checkbox field?..	699
L.5.7 How do I fix a Table/Form showing a wrong numeric-like value in a field? .....	700
L.5.8 How do I fix a Table/Form showing a wrong Short Text value in a field?.	700
L.5.9 How do I fix a Table/Form erroneously ordering records?.....	700
L.5.10 How do I fix a value not in the drop-down menu in a Table/Form field? .....	701
L.5.11 How do I fix a Null in a Table/Form field configured as “Required=Yes”? .....	701
L.5.12 How do I fix an apparent Null in a Table/Form Short Text field configured as “Required=Yes”?.....	701
L.5.13 How do I fix an apparent zero-length string in a Table/Form field configured as “Allow Zero Length=No”?.....	702
L.5.14 How do I fix field value(s) that violate(s) the field/record validation rule(s)? .....	703
L.5.15 How do I fix spontaneously changing Table/Form field values? .....	703
L.6 How do I fix a Table/Form that I cannot open? .....	703
L.6.1 How do I fix “ <i>The record source...does not exist</i> ”? .....	704
L.6.2 How do I fix “ <i>...database engine could not find the object.</i> ”? .....	704
L.6.3 How do I fix “ <i>Could not find file...</i> ”? .....	704
L.6.4 How do I fix “ <i>...is not a valid path</i> ”? .....	705
L.7 How do I fix errors with Short Text or <i>String</i> fields? .....	705
L.7.1 Why text strings can be different from what is shown?.....	706
L.7.2 What unexpected results can I get because text strings are different from what is shown? .....	706
L.7.3 How do I fix text strings and insufficient field width? .....	707
L.7.4 How do I fix text strings and insufficient field height? .....	708
L.7.5 How do I fix text strings and invisible characters? .....	708
L.7.6 What are invisible characters? .....	709
L.7.7 What is the zero-length text string? .....	710
L.7.8 What are invisible text strings? .....	711
L.7.9 Why can I have invisible characters in my Short Text and <i>String</i> fields? ..	711



L.7.10 What apparently defective results can invisible characters produce?.....	711
L.8 How do I fix errors with the user interface? .....	712
L.8.1 How do I fix a missing or disappearing command/tool? .....	713
L.8.2 How do I fix a shaded new-record button? .....	713
L.8.3 How do I fix a Query opening in “SQL View” when I selected “Design View”? .....	713
L.8.4 How do I fix the “Access SQL Editor” marking a Query as having unsaved changes?.....	713
L.8.5 How do I fix an error when opening a Query in “Design View”? .....	714
L.8.6 How do I fix an extra column in my Tables in “Datasheet View”? .....	715
L.8.7 How do I fix an extra row in my Query results?.....	715
L.8.8 How do I fix the database users modifying Table values by editing Query results? .....	715
L.8.9 How do I fix missing columns/rows in my Query results?.....	715
L.8.10 How do I fix “ <i>Could not find field...</i> ” when doing “Compact and Repair” or “Save As”?.....	716
L.8.11 How do I fix the VBA editor changing the value of the constants I write? .....	716
L.8.12 How do I fix foreign-language issues of MS-Access?.....	716
PART M. LIST OF BUILT-IN FUNCTIONS .....	720
M.1 ActiveX functions .....	721
M.2 Application functions.....	721
M.3 Array functions .....	721
M.4 Conversion functions .....	721
M.5 Database functions .....	722
M.6 Date and Time functions.....	722
M.7 Domain Aggregate functions .....	722
M.8 Error Handling functions .....	723
M.9 File Input/Output functions.....	723
M.10 Financial functions.....	723
M.11 Inspection functions .....	723
M.12 Mathematical functions.....	724
M.13 Message functions.....	724
M.14 Miscellaneous functions.....	724
M.15 Program Flow functions.....	725
M.16 SQL aggregate functions.....	725
M.17 File Management functions.....	725

M.18 Text Processing functions ..... 725  
PART N. CONTENTS AND ACKNOWLEDGEMENTS..... I

# Lightning Guide to MS Access & SQL

A practical guide to Database Design  
with MS Access and SQL

